Preface 序

从二十年前个人计算机PC的诞生，BIOS就一直是最底层软件/硬件之间互动的最深层核心，也是每一代计算机平台开发时必须掌握的关键性技术。BIOS的编写更是一门独门工夫，除了以最难写的汇编语言(Assembly Language)编写之外，BIOS的开发必须同时具备支持最新的产业硬件规格，兼顾过去规格(或者说包袱)的兼容性，以及在成本要求下的程序代码最小化的极端要求，至今只有不到四五家软件公司，能够完全掌握这样关键性的技术。

对于BIOS的讨论，绝大多数仅止于BIOS设置值的介绍与讨论，顶多是小小开机画面/图案的小修改，以及将BIOS烧写到ROM的技术讨论；但是若要论及系统内部，去研究BIOS内部的执行流程、支持规格与数据结构等，那么市面上除了二十年前那一本IBM8086/8088 BIOS——业界号称BIOS蓝皮书的中译本外，就没有任何书籍对此作过相关讨论。

因为这种深入系统核心一等一的资料，甚至说程序代码开发的关键性技术，向来是只有IBM、COMPAQ、宏碁、华硕等系统开发厂商才能接触到的机密资料，不是一般人士就能够一窥究竟的。

本书是全球第一本针对新一代计算机平台的BIOS核心技术做深入剖析的技术性书籍，而且本书肯定是全球第一本，内附完整BIOS源代码以及相关技术文献的书籍，完备程度超过二十年前那本IBM BIOS Bible，将这个业界研发技术缺口至少拉近了十八、九年，所以本书的出现，绝对可以说是难得的创举，笔者绝对敢如此夸口！

本书的初步雏形，起自于某种因缘际会，本人与BIOS领导厂商——美商安迈(AMI,American Megatrends Inc.)公司高层人员的接触，该公司鉴于对这种深入系统核心的研究资料在市场上极为欠缺，对于计算机/微电脑

序 Preface

ROM 开发人才的训练与培养绝对是一项缺憾，因此有意将视为商业机密的关键性资料，寻求一个有汇编语言开发背景、对硬件规格了解的专业作者出书。

而笔者近年来较少从事软件的研发，而专注在媒体文章与专栏的发表，以及硬件组装书籍的写作，但十年前也曾经参与防毒软件的研发工作，对于汇编语言与BIOS 内部流程、功能调用等知之甚详，亦或是兴趣、机缘或宿命，于是双方展开密集而漫长的技术合作。

笔者也晓得在当今软件写作的书籍，一窝蜂朝向网页设计，或较为入门的窗口语言写作，从事这种相对冷僻、艰深的书籍创作，真是一项吃力不讨好的工作，但笔者自许为技术入门的带领者，而 AMI 肯将视为商业机密的源代码关键技术完全公开，可见该公司直追 Linus Torvalds 开放 Linux 源代码的开阔胸襟，笔者能够何其荣幸的与该公司充分合作，协力完成本书，为BIOS ROM 人才的培训贡献一点心力；而本人跟 AMI 高层共有的一个梦，一个让计算机人才掌握计算机系统核心级软件技术的美梦下，笔者与 AMI 愿意协力完成，当然也希望读者们能牵成，给予我们支持、鼓励！

本书定位在有基本汇编语言基础，还记得 DOS 操作指令且仍从事软件开发、编程的读者为主。但即便定位如此，本书仍然力求在章节规划、图表以及用字上力求平易与通顺，希望通过这些由浅而深的章节说明，让各位读者对AMI BIOS源代码的内部规划及移植的关键模块 / 源代码等都有了充分的了解；而许多从未在杂志、网络上现身的 AMI BIOS 的工具程序，其功能之强大，在本书详实的操作步骤介绍之下，即使是一般组装玩家，都可以自行尝试而游刃有余！

为了让读者能进一步地体验这种深入BIOS 移植与开发的乐趣，我们特地准备了由 AMI 公司所授权的 BIOS 源代码光盘，这份 BIOS 源代码主要是

Preface 序

针对硅统科技SiS530芯片组结构下所研发的版本，可以移植到各种以SiS530整合芯片组(Socket7 结构，插 Intel Pentium MMX、AMD K6-2 、K6- III以及Cyrix 6x86、6x86MX 等处理器)的主板。读者要花的成本除了本书之外，只要去买一张 SiS530 芯片组结构的主板、Socket 7 结构的 CPU ，一切研发、移植的乐趣，只要自己亲自动手做过一遍，你就能体会！

本书所附的源代码光盘，也许不像 OEM/ 主板合作厂商所付费购买的 BIOS 源代码那样齐备，这是受限于 AMI 美国原厂审核、授权的附赠光盘内容所导致，毕竟有些.ASM 的细节仍然太过敏感，基于避免竞争对手参考或其他市场因素的考虑所致，不得略做些细微的“润饰”。但无论如何，本书所附的源代码光盘内容，绝对可以生成一份可完整烧写、执行的 SiS530 版本的 AMI BIOS，对于读者的实作绝对不会有任何影响，这点是毋庸置疑的，也是笔者跟 AMI 公司交涉过程中所坚持的原则。

最后，笔者也要在此感谢 AMI Taiwan 的 Bird，有了本书的原始构想，以及总经理 S.J Lin 先生的鼎力支持，也感谢吴存展(Den Wu)先生，这位因担任长期接触窗口而日久生情(友情)的新挚友；以及 Brown、Hugh、David、Chung、Pailin 等在写作技术上的全力支持，让我这个患有数年软件开发痴呆的老菜鸟，能够快速归位进入状况；硅统科技(SiS) 产品经理李志村先生，授权 SiS530 技术文献以及相关 Chipset 之 Porting Source，让本书落实一个有真实平台可实作、DIY 的梦想，而不仅仅只是纸上谈兵而已；最后旗标技术编辑志明的全程进度跟踪，在此也一一铭谢！

陈文钦

2001 年 2 月初

关于光盘

说明与需求

本光盘主要内容包含 AMI (American Megatrends Inc.)授权给本书读者参考的 BIOS 源代码(Source Code)，版本为 6.27.10 (简称 627.10)。

由于本 Open Source BIOS 源代码是针对采用 SiS530 整合芯片组为结构的主板而设计的，所以当你准备钻研此源代码，或编译自己的 BIOS 监控程序之前， 需要自行准备一些工具才行。在硬件部分，请准备一片 SiS530 (不限品牌) 的主板，与足够的 Flash 工具 (如烧写 BIOS 用的 FlashROM、拔 IC 专用的 IC 夹等等)，以作为研究的材料。

在软件部分，还需要下列编译 / 工具程序文件：

文件名称	建议版本	用途说明
NMAKE.EXE	1.40	程序开发维护工具(保护模式可执行文件)
NMAKER.EXE	1.30	程序开发维护工具(保护模式可执行文件)
MASM.EXE	6.11	Macro Assembler 6.11 主编译程序
ML.EXE	6.11	Macro Assembler 6.11 后勤编译程序兼连结程序 (Linker)
LIB.EXE	3.20	Library Manager 链接库删减 / 编修工具程序
LINK.EXE	5.31	目标码(Obj) / 链接库连结器(Segmented Linker)

这些文件隶属于 Microsoft Assembler 6.11，或者在 Visual Studio 6.0、Visual C++ 、MSDN(Microsoft Development Network) 等微软程序开发套件中可以找到。为了维护智能财产权，这些编译程序请读者自行备妥。至于详情，请看第 4 章的说明。

关于光盘

文件安装

安装方式非常简单，假设你光驱代号为 R:， 只要在 DOS 下 (或者在 Windows 95/98 的 MS-DOS 窗口下)，输入：

```
XCOPY R:\627.10 C:\627.10 /S
```

就可以将所有 BIOS Source 拷到 C:\627.10 。当然也可以在 Windows 资源管理器中用拖曳的方式将文件复制到 C:，但请记得事后用 ATTRIB -R C:\627.10 /S 的指令将所有文件的只读属性去除，以免编译时发生错误！

环境设置

大多数程序语言编译器都需要设置编译环境，本 BIOS 也不例外。这里的 BIOS 程序代码几乎是以汇编语言(Assembly)来设计的，须通过 MASM 6.11 来编译。编译时请记得设置以下的环境变量，假设 MASM 6.11 是安装到 C:\MASM611，则需要作以下的设定：

```
set include=c:\masm611\include\
set helpfiles=c:\masm611\help\*.hlp
set init=c:\masm611\init\
set asmex=c:\masm611\samples\
set tmp=c:\masm611\tmp
set demo=c:\masm611\demo
```

在 627.10\EXE 目录下有个简单的批处理文件 Setpath.bat，只要执行过一次，就会设置好上述相关的路径，你也可以在 AUTOEXEC.BAT 加入执行 setpath.bat 的叙述。

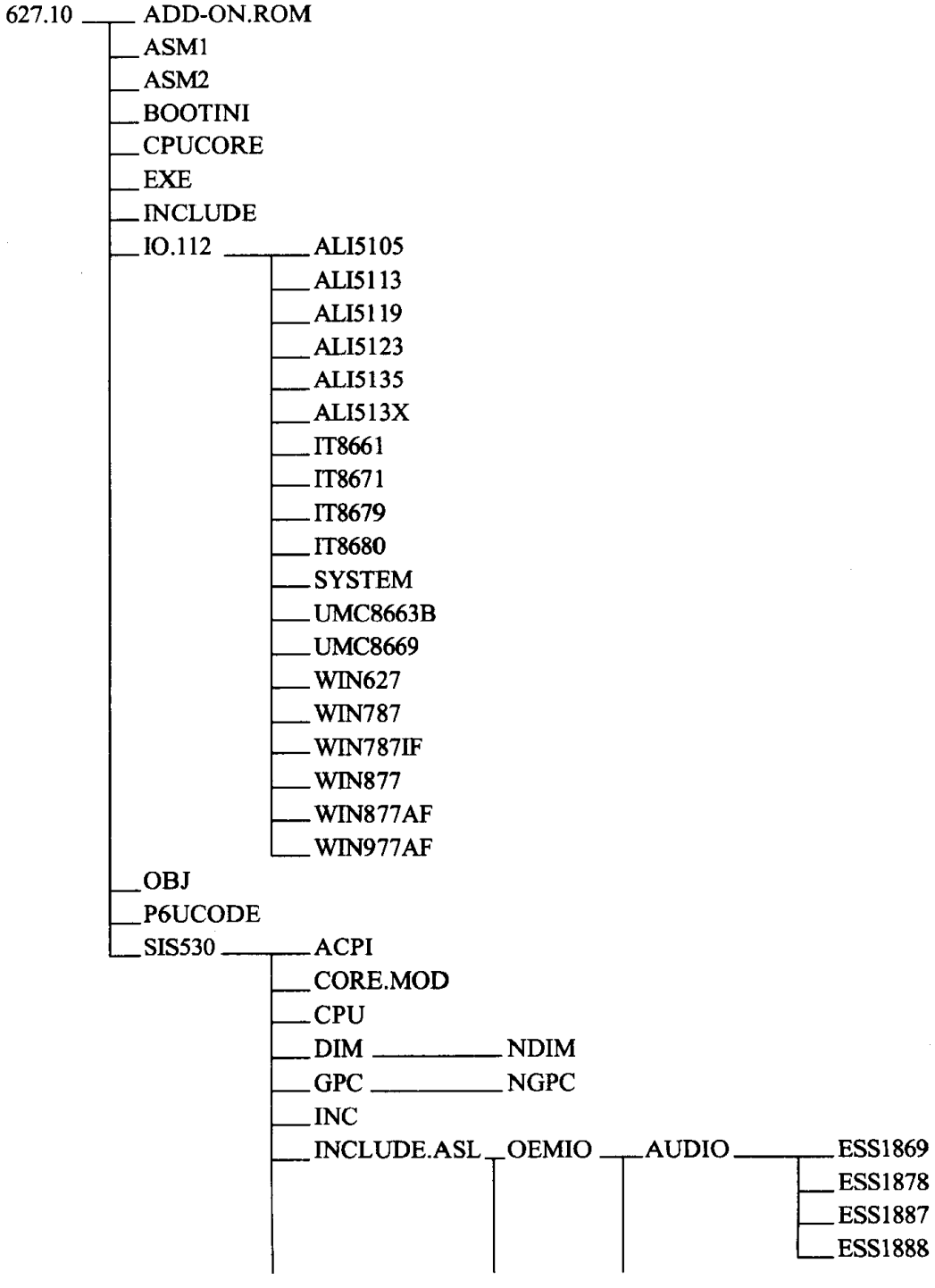
其他

至于其他事项、内容，请参照本书说明。

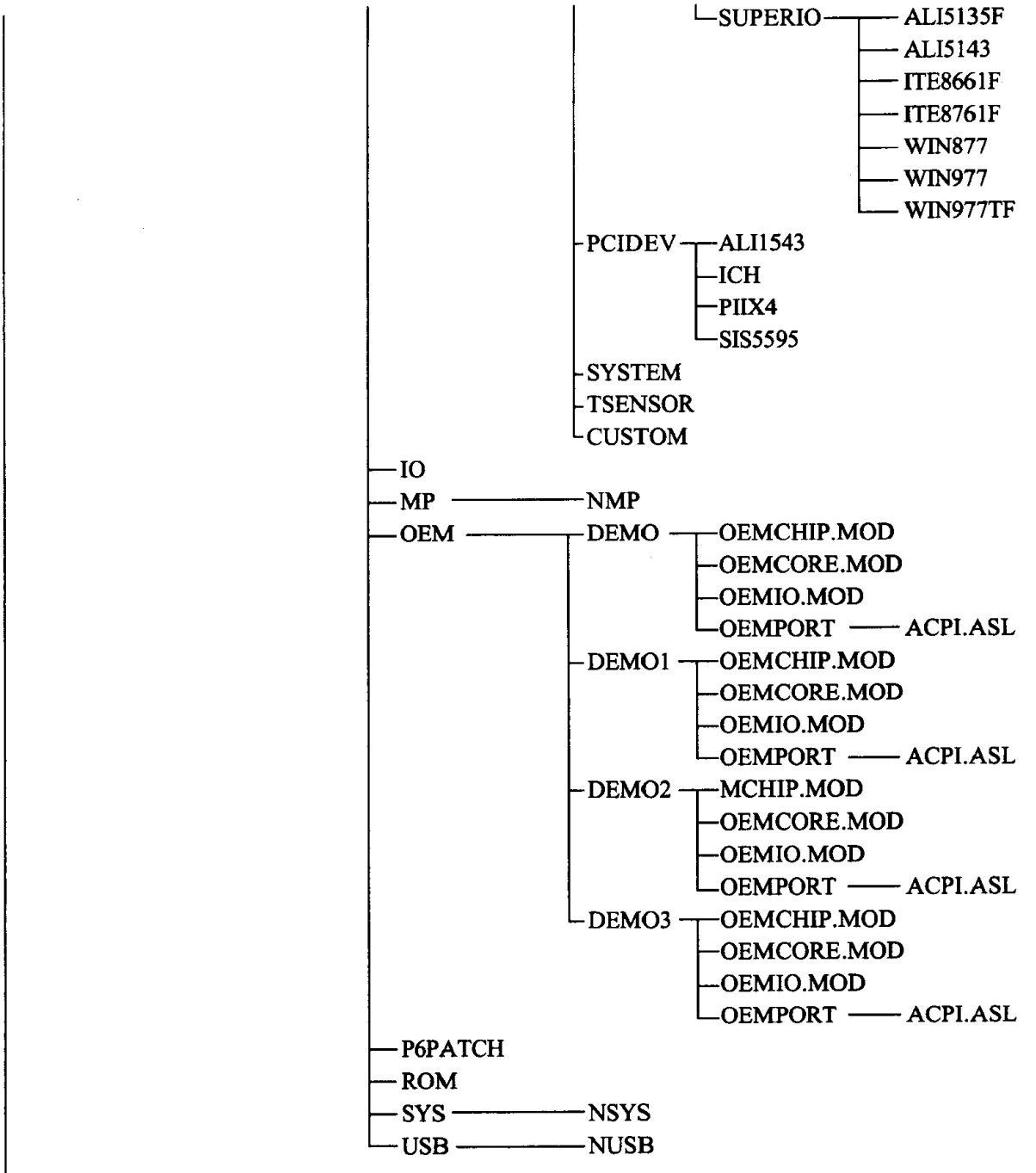
关于光盘

目录结构

本光盘的目录结构如下：



关于光盘



以上各目录的用途，在本书第 4 章都有详细的介绍，请自行参考本书内的说明。

Contents 目录

第 1 章 BIOS 基本概论与设置

1-1 BIOS 的定义与解释 2
1-2 何谓 BIOS 设置、如何更改 CMOS 设置 8
1-3 BIOS 市场风云录 15
1-4 简介美国安迈(AMI,American Megatrends Inc.)公司 21

第 2 章 硬件基本概论与规格导读

2-1 计算机处理器、主板、芯片组的介绍 30
2-2 主板零组件、总线等特定规格介绍 44
2-3 跟电源管理有关的规格与术语解释 58

第 3 章 BIOS 开发与硬件之间的互动关系

3-1 CPU 历代演变与 BIOS 的关系 70
3-2 x86CPU 内存结构与 ROM 大小的演进 74
3-3 BIOS 的诊断码 (Post Code) 简介 87
3-4 BIOS 所要掌握的系统资源 96
3-5 CMOS 的索引数据 103

第 4 章 AMI 源代码目录结构与文件维护

4-1 AMI BIOS 的源代码版本缘由 114
4-2 AMI BIOS 的源代码目录设计 115
4-3 编译 BIOS 所需要的编译 / 链接程序 125
4-4 BIOS 的编译 / 维护作业的相关文件之一 (IO.MAK) 129
4-5 BIOS 的编译 / 维护作业的相关文件之二
——供 OEM 厂商自定规格 / 功能等设置值的维护
文件OEM.MAK 138

目录 Contents

第 5 章 BIOS 的内部模块与执行流程介绍

5-1 BIOS 内部模块的说明 160
5-2 BIOS 执行流程图 165
5-3 BIOS 程序源代码浏览(一)
——BootBlock(激活区块) 的相关模块.....180
5-4 BIOS 程序源代码浏览(二)
——Flashpart(快速存取存储器) 相关程序代码分析.....202
5-5 BIOS 程序源代码浏览(三)
——其他在 POST 阶段参考到的程序代码分析.....212

第 6 章 BIOS 平台移植的关键技术——特定平台修改与关键模块 / 源代码的介绍

6-1 掌握芯片组、外围 I/O 芯片的相关规格 226
6-2 移植到特定芯片组要修改的源代码 240
6-3 内存容量检测 / 计算 247
6-4 PCI IRQ 号码配置与绕线设置 279

第 7 章 BIOS 画面选项、默认值设计

7-1 AMI BIOS 的开机设置画面与选项简介 294
7-2 以 AMIBCP 调整开机默认值 / 隐蔽项目 301
7-3 AMI BIOS 的 Setup Script (.SSP) 语言的介绍 306
7-4 直接取用源代码文件群的.SSP 来修改 338

Contents 目录

第 8 章 AMI BIOS 工具程序——ROM 升级与编辑工具

8-1 AMI BIOS 基本工具程序列表 342
8-2 先从烧写 BIOS 开始
——强大的 AMIFLASH ROM烧写程序 346
8-3 BIOS 组态魔术师——AMIBCP.EXE 之一 369
8-4 BIOS 组态魔术师——AMIBCP.EXE 之二 380

第 9 章 AMI BIOS 工具程序——研发 / 维护工具

9-1 BIOS 组态魔术师——AMIBCP.EXE 之三 388
9-2 AMIMM.EXE BIOS 模块操作 / 修改程序 398
9-3 AMIMM.EXE 的描述 / 操作语言介绍 400
9-4 RU.EXE 硬件系统寄存器检视程序 407
9-5 其他研发工程师 / 高等玩家的工具程序 417

第 10 章 硬件移植实战 DIY

10-1 简介本书所示范的主板与其他配备 431
10-2 建立必要的目录、相关文件与路径 437
10-3 开始进行 Porting 作业 441
10-4 修改开机的设置界面、商标图案与全屏幕画面 456
10-5 BIOS 开发的错误解决与明日应用 476

附录 A 483

CHAPTER

1

BIOS基本概论与设置

1-1 BIOS 的定义与解释

大约在二十年 (1981) 前，当时被信息界称为蓝色巨人的 IBM，在研究第一部个人计算机——IBM PC 时，他们的工程师把一些开机时的硬件启动 / 检测码 (Initial Code)，从软盘或硬盘加载到操作系统，将开机程序的前导程序代码，以及一些最基本的外围 I/O 处理的子程序码 (如屏幕显示、磁盘驱动器驱动、摇杆控制等)，通通挤入在一块大约 32KB 大小的 PROM (Programmable ROM，可编程只读存储器)中。这个程序代码就叫做 BIOS (Basic Input/Output System)，如图 1-1 所示计算机主板中的 BIOS 芯片。

当你的计算机从开机的一瞬间，硬件特性就由 CPU 从主板的 BIOS 芯片内取得程序代码，用 BIOS 内部的程序代码获得控制权并且发挥作用，从 CPU 内外部的检测设置、激活 DRAM以及针对芯片组与各种外围设备作初始化设置之后，最后驱动软盘或硬盘，直到把操作系统 (如 DOS，Win98/NT，Linux) 加载成功，BIOS 的开机引导工作就此告一段落，转向从事幕后的支持、协调工作，并帮助操作系统或应用程序，来处理外围设备沟通的细部操作。

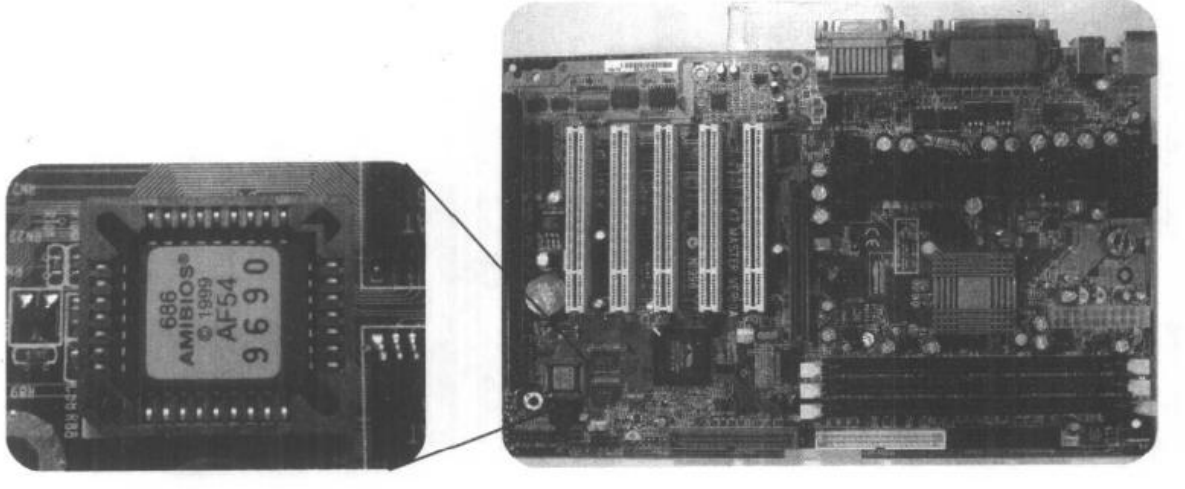


图1-1　唤醒整个计算机系统的 BIOS 芯片

事实上除了主板上有 BIOS 之外，另外像显卡、SCSI 控制卡、磁盘附加卡或高级网卡也都有自己的 BIOS 芯片。这些设计在主板上、负责整个主板运行的 BIOS，我们一般称之为 System BIOS 或 Mainboard BIOS；显卡上的 BIOS 多半以 Video BIOS 称呼，SCSI 控制卡的 BIOS 则以 SCSI BIOS 称呼。

由于一般人所接触的计算机系统，最起码有一组 Mainboard BIOS，而且有些主板已经整合了显卡、SCSI 接口，因此 Mainboard BIOS 已经涵盖了以往 Video BIOS 与 SCSI BIOS 等监控程序。为了方便本书的阅读，如果没有特别声明的话，BIOS 就是指主板上的 BIOS。

如果要更清晰地定义什么是 BIOS，在一家 BIOS 研发厂商的技术培训文件中，对于 BIOS 的叙述，他们是这么说明的：

> A Firmware program (一种属于 ROM 的程序代码)，存储在 ROM、EPROM 或 FlashROM 内存中，提供最基本的硬件初始化 (initialize)、外围控制的必要程序代码。通常是用汇编语言 (Assembly Language)编写的。

BIOS 平时设置为只读的状态，除了特殊工具程序或特殊方式之外，任何应用程序都没办法对它进行修改或删除。以前的修改方式比较复杂，如果是 Mask-ROM 或 PROM，几乎得重新换一块空白的 ROM 芯片，然后烧写。但到了 EPROM 时代，可以用紫外线照射其窗口一段时间，在重新擦除其内部的所有数据之后，再用 EPROM 烧写器将 BIOS 数据烧写到 EPROM 中。

当 Windows 95 流行之后，由于 Plug and Play (PnP) 与 ACPI以及DMI 协议的需要，主板中需要有一个可以记录外围设备资源的异动情况，却又不会被其他程序所破坏的内存区域，而FlashROM 对于ROM程序代码的更新比 EPROM 还要方便。所以到后来，从主板 BIOS，以至于显卡、各外围接口卡存储 BIOS 的内存芯片，几乎都是 Flash ROM 的天下。

一般在设计计算机或外围设备时，存储 BIOS 程序代码所使用的只读存储器 (ROM) 种类，大致可以分为下列几种：

① PROM (Programmable ROM) 可编程存储器

PROM 出厂时内部每一字节的数据都是$FF (也就是每个位都为 1)，且从未烧写过，可供客户用烧写器以特殊电压将数据写入 (把某个位写入 0，就相当于把代表 1 的线路给烧断) 。但是也因为这种特性，PROM 只能写入一次数据，日后要再次修正数据时，必须重新购买一块新的空白 PROM 来烧写。

② Mask ROM

它是由客户指定的数量较大的，由内存生产厂家依客户的指定内容，在大量生产的过程中直接将数据写入，出厂之后就已经有特定的程序 / 数据码，内容也无法自行修改或烧写。通常在大量生产下， MaskROM 的容量比较大，因此成本也比较低，这也就是早期许多汉卡字形、甚至大型电子游戏机喜欢采用 MaskROM 的原因。

③ EPROM (Erasable PROM)

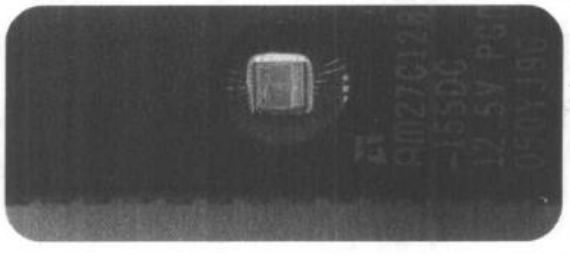


图1-2 EPROM 有个窗口

接下页

这是从 PROM 之后的改良型 ROM 。EPROM 有个直透芯片内部的透明的圆形窗口(如图 1-2 所示)。若要清除数据，可以用专用的 EPROM 烧写器，或者紫外线灯管照射几十分钟后，内部数据就重新恢复成 $FF (位恢复成 1)，用户可以重新烧写新的程序 / 数据码，而不必再去购买一块芯片来烧写，对厂商与客户在监控程序维护上比较方便而且经济。

在 Flash 内存出现之前，EPROM 一直是主板、显卡或其他的外围设备所常采用存储 BIOS 监控程序的内存类型，但随着写入速度更快、更方便的 Flash 内存的出现，EPROM 逐渐被取代。因为并不是每个客户都有 EPROM 烧写器，也不愿意大费周折花几十分钟先清除 EPROM 的内容，然后再烧写数据。

④ Flash ROM

Flash ROM 快速只读存储器 (又有人称 Flash Memory)，简称 FlashROM，是当今主板、显卡存储 BIOS 监控程序的主要内存类型。许多数码相机、PDA 以至于大哥大手机等便携式设备，都使用这种类型的内存，因此 FlashROM 可说是目前最当红的内存种类之一。

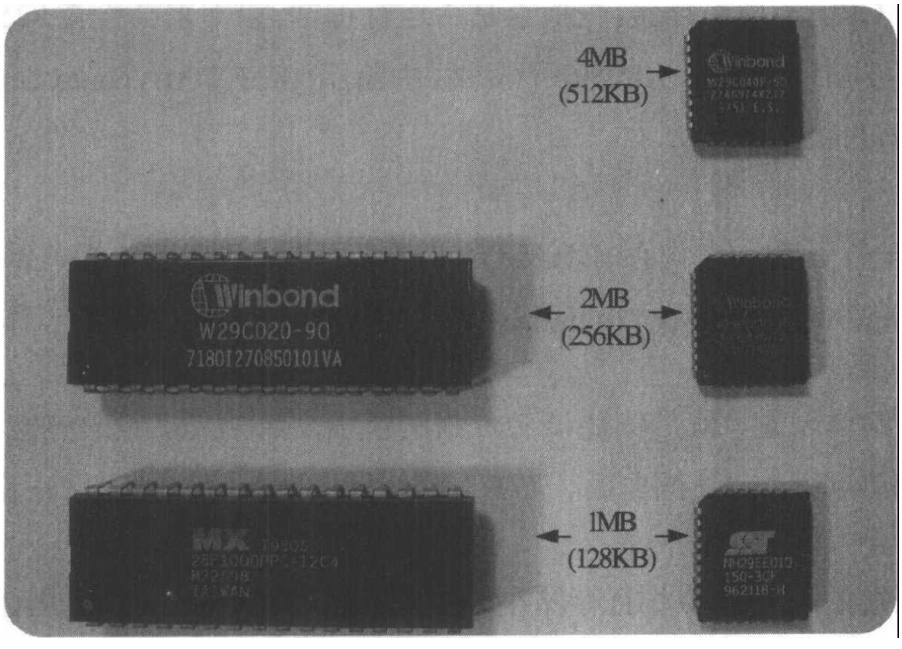


图1-3　1MB，2MB 与 4MB 大小的各种 FlashROM 芯片

接下页

这种内存在一般状态下，特性和一般的 ROM 没什么两样，也就是先前写入的数据无法被擦除、破坏，即使关掉电源之后内容仍旧存在；不过在特殊工具程序的配合以及相关硬件设置、某种特定电压之下，内部存储的程序代码、数据，能够快速地被擦除，并且可以重新修改、写入新版本的监控程序与数据。

FlashROM 读取的最小单位是一个 byte，但擦除 / 写入的单位是以一个“区块(Block)”来区分的，大小从 4K、8K 到 16K 不等。以一个最小为 4K写入单位的 FlashROM 而言，即使你要修改某一个 byte 的数据，必须得将该区块的所有内容备份出来，改变其中要修改的部分后，再一起把整个区块写回去。

以读者最常接触到的主板 BIOS 而言，它最重要的工作有下列几点：

① POST (Power On Self Test, 开机自测试) 一开机系统将控制权交给BIOS时, 它会针对 CPU 各项寄存器，先检查是否运行正常，接下来会检查 8254 timer (可编程外围计时芯片)、8259A (可编程中断器)、8237DMA controller (DMA 控制器)的状态。

② Initial—— 针对动态内存 (DRAM) 、主板芯片组、显卡以及相关外围的寄存器 (register) 做初始化 (Initialize) 设置，并检测是否能够正常工作。

所谓初始化设置，就是依照该芯片组的技术文件规定，做一些寄存器填值、改位的动作，使得主板 / 芯片组的内存、I/O 的功能得以正常运行。

③ 记录系统的设置值，并且存储在非挥发性内存 (Non-Volatile RAM)，像 CMOS 或 Flash Memory (ESCD 区域)等 。

④ 将常驻程序库(Runtime Program)常驻于某一段内存中，提供给操作系统或应用程序调用，像 Int 10h、Int 13h、Int 15h 之类的函数。

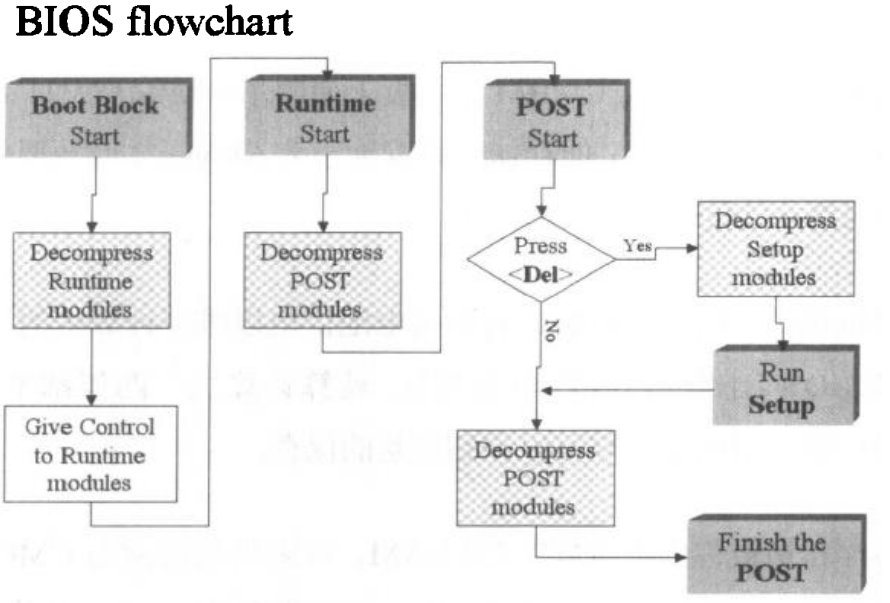


图 1-4 BIOS 开机时的简易流程图

以上几个动作，很像飞机驾驶员在坐进飞机之前，先对飞机整体做一番巡视，做大概的检查 (Post)，然后按一定的流程，逐一打开各仪表板上的开关与灯号 (Initial)，接下来就按航空日记或行程表，设置好飞行的坐标，然后飞机就可以准备起飞了(Boot)。

而在起飞之后，Runtime Program 就像是隐藏在仪表板后方、正在运行的控制系统，这也是飞行员要留心的部分。希望这样的比喻，能让你体会 BIOS 千头万绪的操作流程。

事实上这些只是比较简要地叙述了 BIOS 开机的主要步骤与流程，在以后几个章节中，我们将会进一步剖析 BIOS 的数据结构、变量定义，以及一些重要内部模块的介绍。

1-2 何谓 BIOS 设置、如何更改 CMOS 设置

CMOS是 (Complementary Metal Oxide Semiconductor)，它原指一种 IC 的制程 (IC 分 TTL、CMOS、BiCMOS 等)，用 CMOS 制作的 IC 具有省电、低温的特性。最早在 286 (AT) 时代，IBM 在主板上使用了一块编号 MC 146818 的 CMOS 参数记录 / 计时芯片，能存储的数据大小才 64bytes，而且要扣掉一些固定空间来记录系统日期 / 时间的字段。

当计算机电源关闭时，这块 IC 会改由电池供电以维持计算机的设置记录值。由于 CMOS 制作的内存耗电量很低，就算计算机一两年都不开电源，CMOS 内存内的记录值也一样可以得到完整的保存。

这种芯片起初被称为 RTC/CMOS RAM，后来则被简称为 CMOS 芯片。随着 PC 工业的蓬勃发展，也有厂商设计出不同类型的 CMOS 芯片。记得早期的主板，CMOS 芯片边有一个像啤酒桶般的电池，而这种设计往往随着时间的长久，电池内部的电解质液体的流出，造成主板电路被腐蚀。随后也有厂商使用一种整合电池与 CMOS RAM 于一体的芯片，它被设计成 IC 插座的类型，但体积 (或者说厚度) 比较大，这类整合电池的 CMOS 芯片也曾流行一时，如图 1-5 所示。

图1-5 将电池整合进去的CMOS 芯片

随后现代主板的线路逐渐整合、简化，这种 RTC/CMOS RAM 硬件结构逐渐被保留并整合到像 Super I/O 芯片甚至南桥芯片 (South Bridge) 等，内容也不再只有 64bytes (可能到 128bytes，甚至更多)；而提供电源的电路则拉到芯片外头，另外搭配锂电池来供电，旁边仅保留了 CMOS 放电的 Jumper，供厂商或客户清除 CMOS 记录值用，如图 1-6 所示。

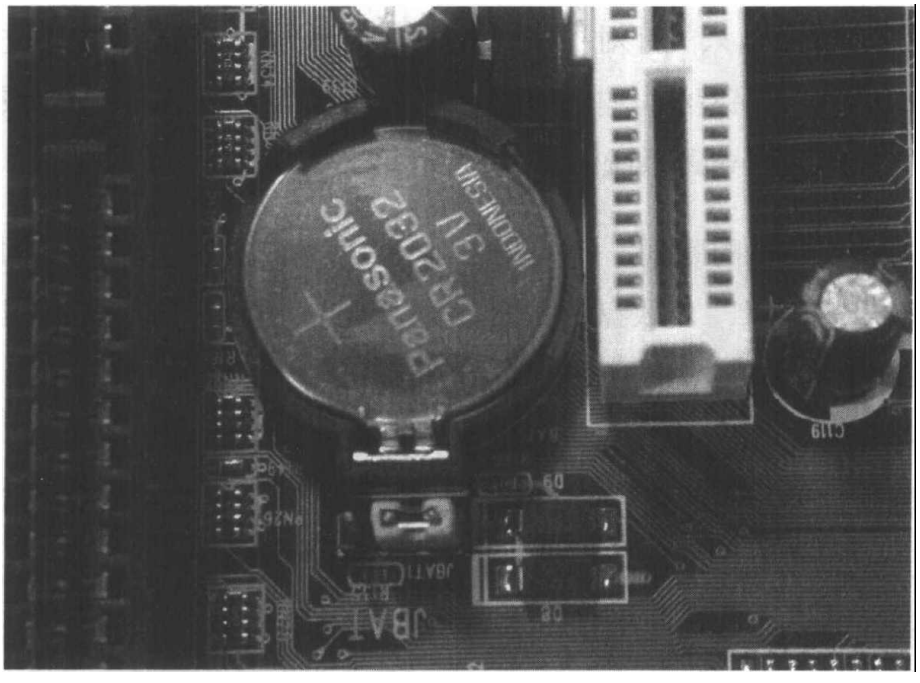


图1-6　今日的主板除了锂电池与 CMOS 放电 Jumper 外，已看不到 CMOS 芯片的踪迹

所以现代的主板中，除了电池与 CMOS 放电的 Jumper 以外，已经看不到所谓的 CMOS 芯片。对于 BIOS 而言，它依旧存在；但对于主板或客户而言，它已经成为南桥芯片的一部分。

那到底什么是BIOS设置或CMOS设置呢？

随着当今主板的功能包罗万象，支持的CPU种类、频率／电压设置的多元化，外围设备五花八门，加上DIY的日见兴盛，买过计算机的用户都会或多或少地购买一些外围设备、内存、硬盘或显卡来扩充、加装、升级，虽然BIOS已经尽可能地去做一些侦测外围设备的动作，但是它不可能将所有的外围设备都照顾到，只能针对一些开机时马上就要用到的外围设备先进行处理，而有些功能则是保留给某些设备装上时才启用的，若贸然启用会减慢系统开机的速度，甚至会导致系统组件之间的冲突。

因此，除了开机的瞬间，中央处理器(CPU)的品牌、型号与频率，以及内存的型号、容量，及软盘、硬盘、光驱(CD-ROM)等基本外围设备的侦测与驱动之外，基本上，BIOS将这些零零碎碎的大小设置项目，根据设置属性、用途大致分成约十几个大项目，用户先选择大项目之后，再进到这个项目去调整每个参数的设置值，这些参数值会存储并记录在CMOS芯片内，在下一次开机之前，由BIOS依照用户的设置需要，来激活／关闭某项功能，或者侦测某项设备，这就是所谓的BIOS或CMOS设置。

以美国生产的安迈 AMI BIOS 为例，在你一打开电源时，在屏幕刚显示一开机 EPA 能源之星的环保图案 时，立刻按下 [redacted] 键，就可以进入 BIOS 的系统的设置画面，如图 1-7 所示。

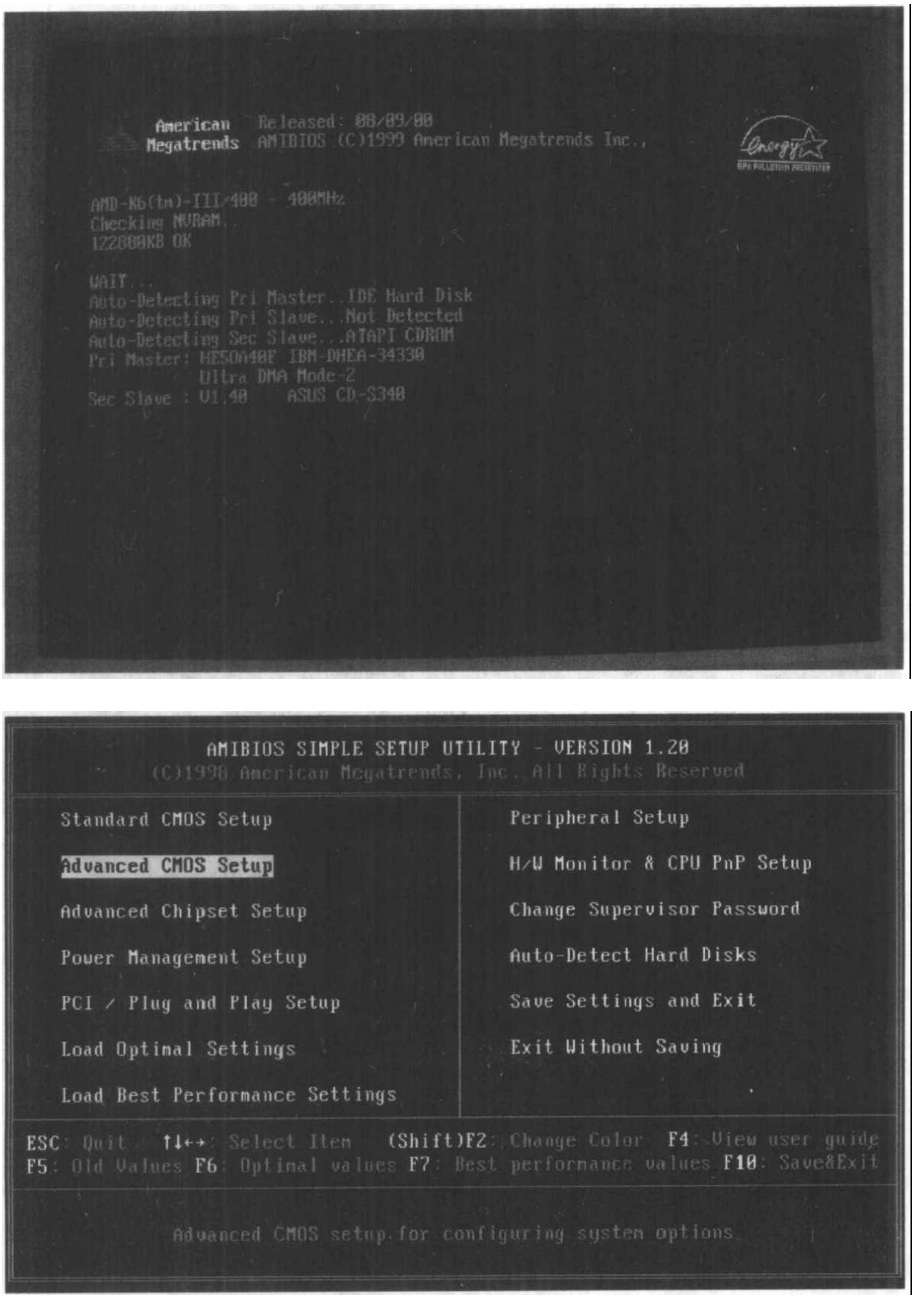


图1-7　进入 BIOS 设置画面，此为 AMI Simple Setup 的 BIOS 画面

至于在一般计算机之中主板较为常见的 AWARD BIOS，也同样是在开机画面出现时，赶快按下 [redacted] 键就可进入 BIOS 的系统设置画面，如图 1-8 所示。

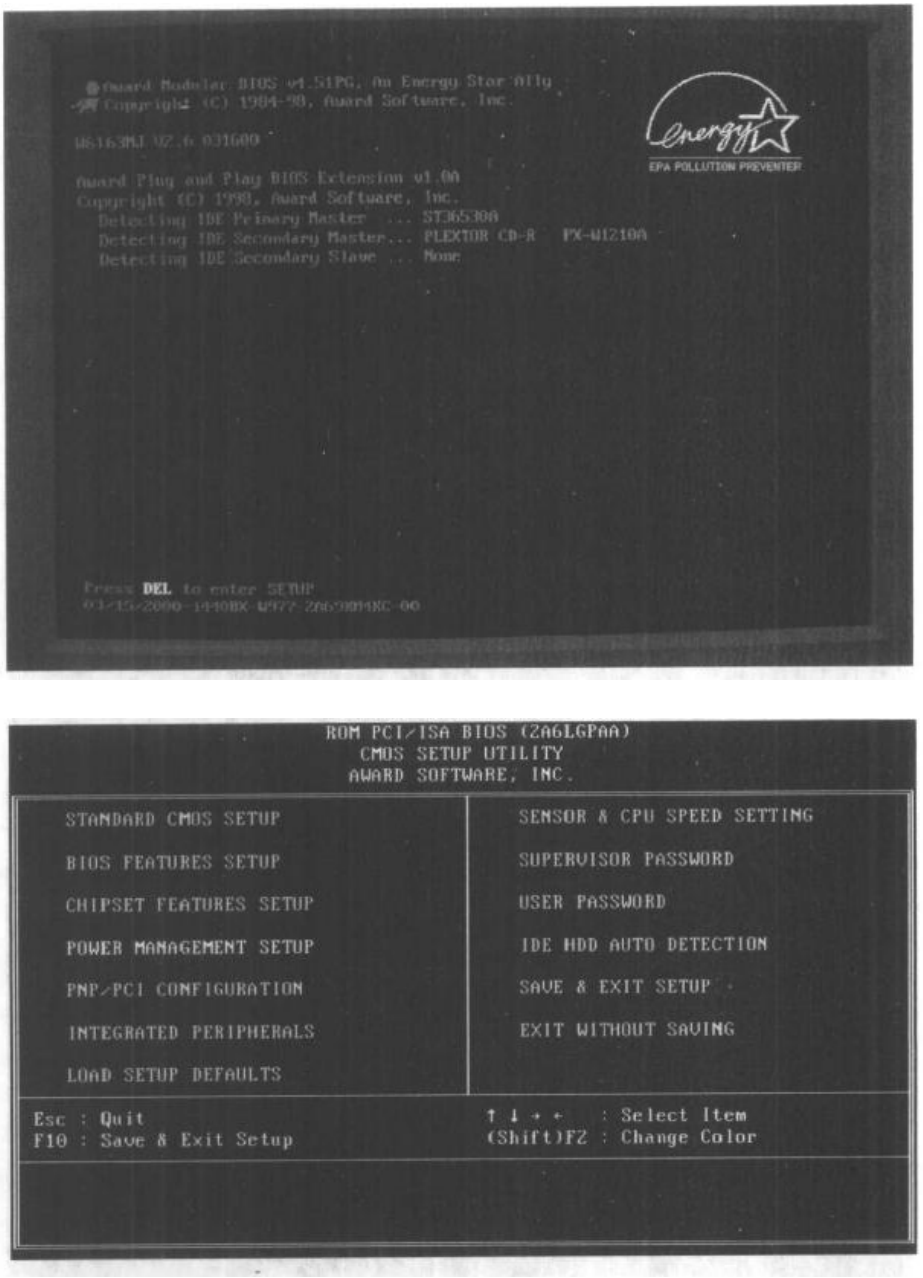
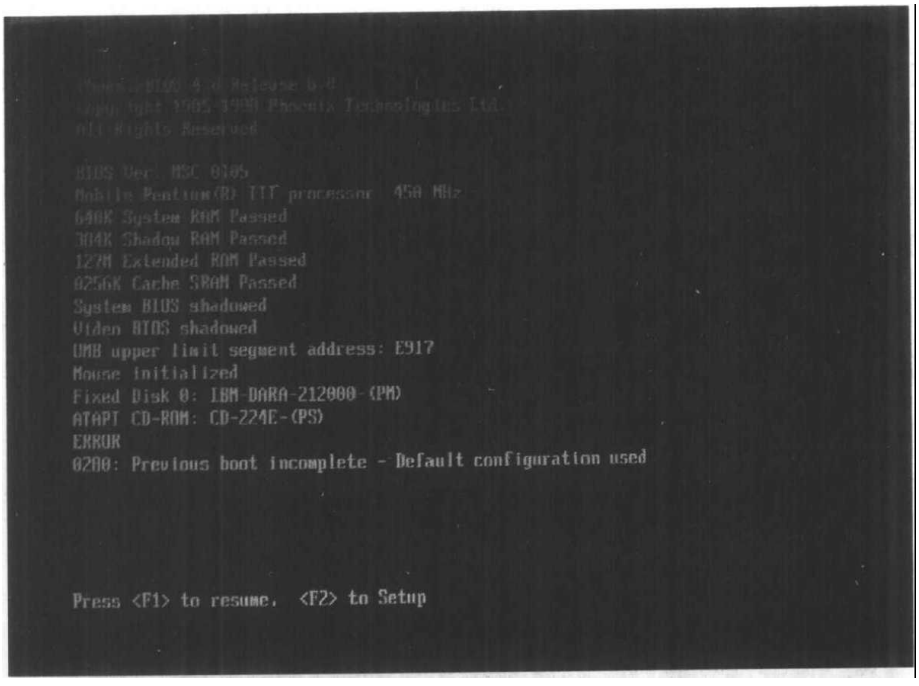


图1-8　主板较常见的 AWARD BIOS 设置画面

而在笔记本电脑中较为常见的是 Phoenix BIOS，当开机时按下 [redacted] 键，则会进入 BIOS 的画面，如图 1-9 所示。



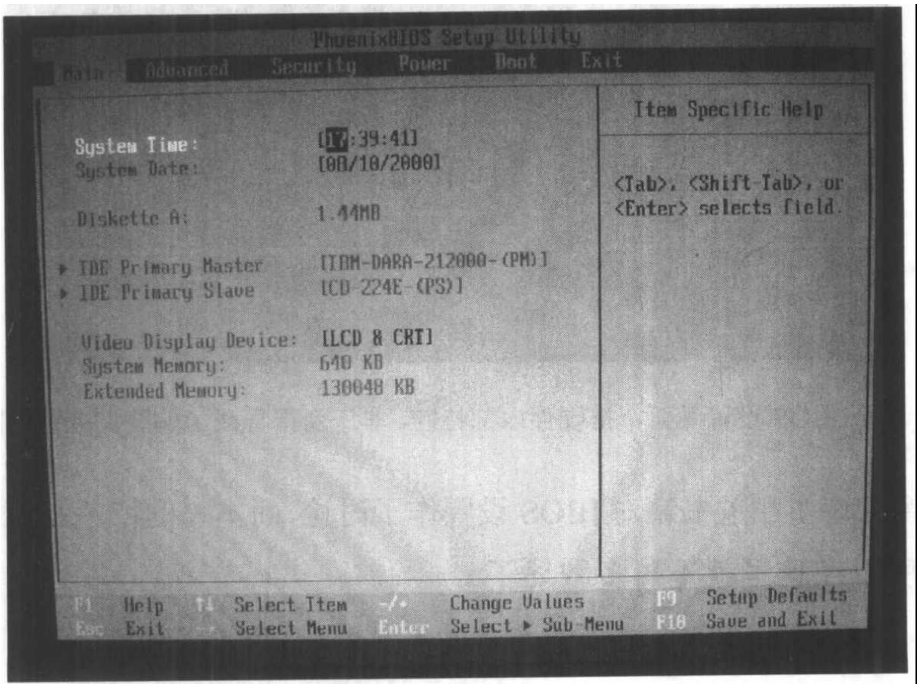


图1-9 Phoenix BIOS 的设置画面

在大多数 BIOS 设置项目中，有些要记录在 CMOS 的参数值只是日期、时间的设置，调整这些参数就跟玩电子表一样的简单；而有些功能选项，就属于进阶的性质，你需要对计算机外围、零组件有基本的了解 (最起码要懂得 LPT1、COM2、on board IDE 等术语)，如果你是爱好自行组装 (DIY) 计算机的好手，这些选项应该难不倒你了。

不过还有些 BIOS 参数值，涉及到硬件的特性、局限度等，这些参数依照 CPU、芯片组，甚至跟主板的设计、使用的零组件都有密切的关系。在出厂之前，厂商已经帮我们调整好最稳定且适当的参数值了(如图1-10所示)，一般而言我们都是按照厂商的默认值而不要去改动它，这些琐碎繁杂的硬件选项要是不懂而胡乱设置的话，可能会导致计算机速度变慢、运行异常，甚至无法开机。

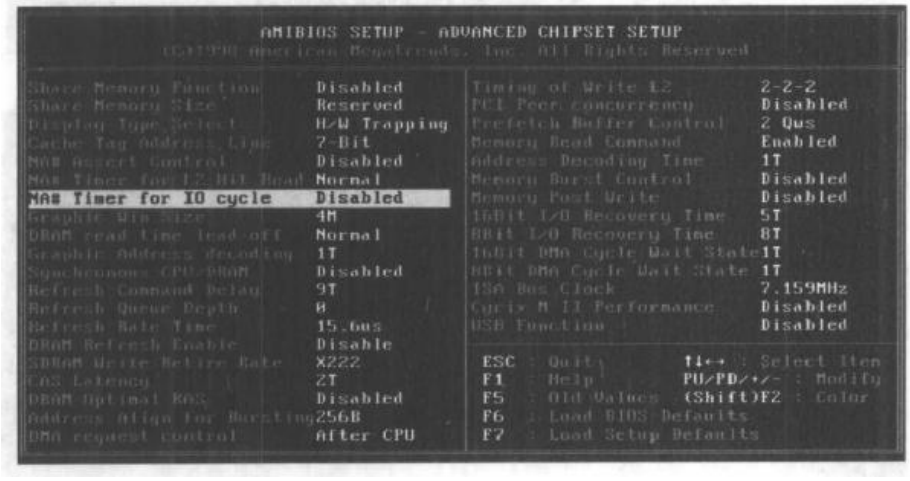


图1-10　部分 BIOS 设置画面牵扯硬件内部特性，非厂商指导或了解硬件者请勿随意设置

已经有许多相关书籍对 BIOS 设置作了讨论，而本书的特色是以技术剖析为主，所以在这里就不再多叙述了。

1-3 BIOS 市场风云录

在二十年前，IBM 推出个人计算机时，其内部 64KB 大小的 BIOS，完全由 IBM 工程师自行独立开发，当时为了快速的推广并普及，IBM 将 PC 的内部系统等技术资料，甚至 BIOS 系统调用等接口规格都予以公开，以协助其他厂商开发 PC 的相关软件、硬件产品。

后来像 COMPAQ 等计算机厂商，为了要开发跟 IBM PC 兼容的个人计算机，在维持相同的函数调用界面，在与操作系统、应用程序兼容的前提下，尝试自行开发 BIOS。当时市面上有一本专门剖析 IBM PC、PC XT BIOS 原始机器码的参考书，而这样的技术宝典，也是早先最早研发个人计算机厂商技术人员的必备工具书。

一些最早踏入个人计算机研发领域的厂商如宏碁、神通，以前也是投入大量人力来开发、编写自己计算机的 BIOS，除此之外其他厂商，在当年也可能游走在法律边缘，以所谓的逆向工程还原技术 (Reverse Engineering) 去直接反编译一些厂商生产的 BIOS 来使用，这样做当然是 100% 的兼容，但是很容易遭到许多法律诉讼。由于涉及到专利权与厂商之间的竞争因素，BIOS 的兼容性与合法性，一直是 PC 界要面对的严肃课题。

在 1983 年 XT 时代，工研院电子工业研究所 (ERSO)，曾以 Clean Room (纯净室) 的方式，开发出与 IBM PC 兼容的 ERSO BIOS，并授权转移给一些厂商使用，也因此打开了一些由大厂掌握 BIOS 关键技术的封闭缺口；到了 286 时代，美国凤凰科技 Phoenix Technologies 也以同样的手法，开发出功能强大且兼容的 286 AT BIOS，并且以授权的方式，提供给一些有意生产个人计算机的厂商使用，而厂商也可以通过 BIOS 厂商所提供的 BIOS 源代码与相关的技术支持，快速地开发出新一代的计算机产品并推出上市。

随后，又有美国的安迈公司 (AMI,American Megatrends Inc.)，Award Software Inc (惟尔科技)，SystemSoft 等软件公司也加入到角逐与经营中。至此，BIOS 的研发也成为了一个专门、有技术门槛的软件行业。

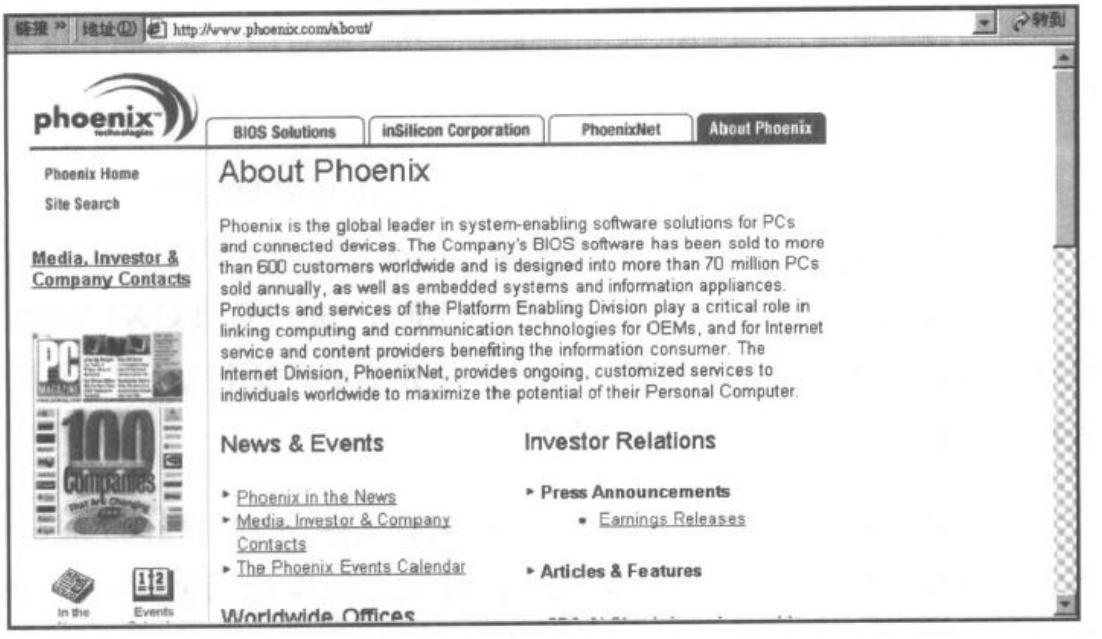


图1-11　Phoenix 合并 AWARD 后声势大涨，是目前 BIOS 的龙头

目前从事 BIOS 程序代码研发的公司有 American Megatrends, Inc. (AMI, 美国安迈)，Phoenix Technologies (美国凤凰科技)，Award (惟尔科技，后来为 Phoenix 于 1998 年9月所并购)，无论是以研发根基深厚、开机速度快捷闻名的 AMI BIOS，还是台式机 (尤其主板) 较为常见的 AWARD BIOS，以及笔记本电脑中最常见的 Phoenix BIOS，这三家 BIOS 系统软件公司的 BIOS，在全球 BIOS 占有率绝对是最高的，可说是当今 BIOS 一等一的大型厂商。

除此之外，在笔记本电脑市场中，原本还有一家为美国 SystemSoft 的公司，该公司的笔记本型 BIOS 也极为有名，其市场占有率紧追着 Phoenix NoteBIOS，但随后又有一家于 1998 年9月成立的系微股份公司 (Insyde的网站如图 1-12 所示)，买下整家 SystemSoft 公司 BIOS 版权与相关研发部门，随后引进处理器龙头企业英特尔 (Intel) 等机构的资金，以当地的 BIOS 专业厂商为号召，其业务方向着重于 OEM 主板市场，特别是笔记本 / 便携式计算机系统的 BIOS 市场。

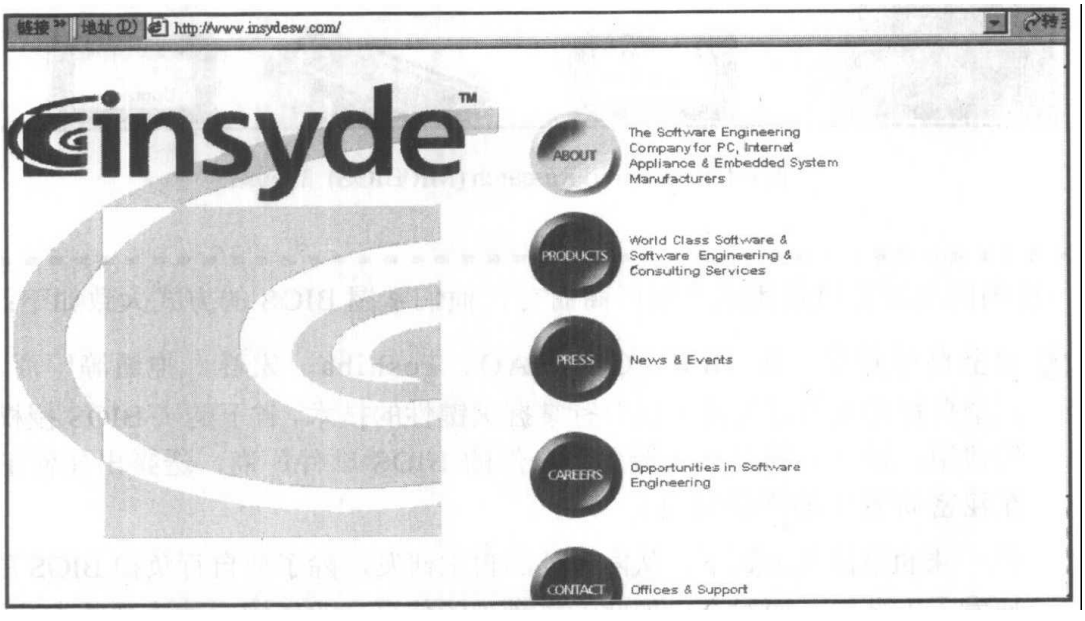


图1-12　系微 (Insyde) 公司的网站

另外在台式机市场部分，在两三年前有一家 Microid Research Inc. 公司(如图 1-13 是该公司的网站)，以 Shareware (试用推广) 的方式，开放让客户先自行下载 BIOS 程序来烧写、升级，以延伸、扩充自己的主板功能，试用满意后再付款，此举也使该公司的 MRBIOS出了名；但随着被 Unicore Software 公司买下后，以 Shareware 先行试用的策略大转变，MRBIOS 不再开放给用户下载，MRBIOS 也从此归于平淡。

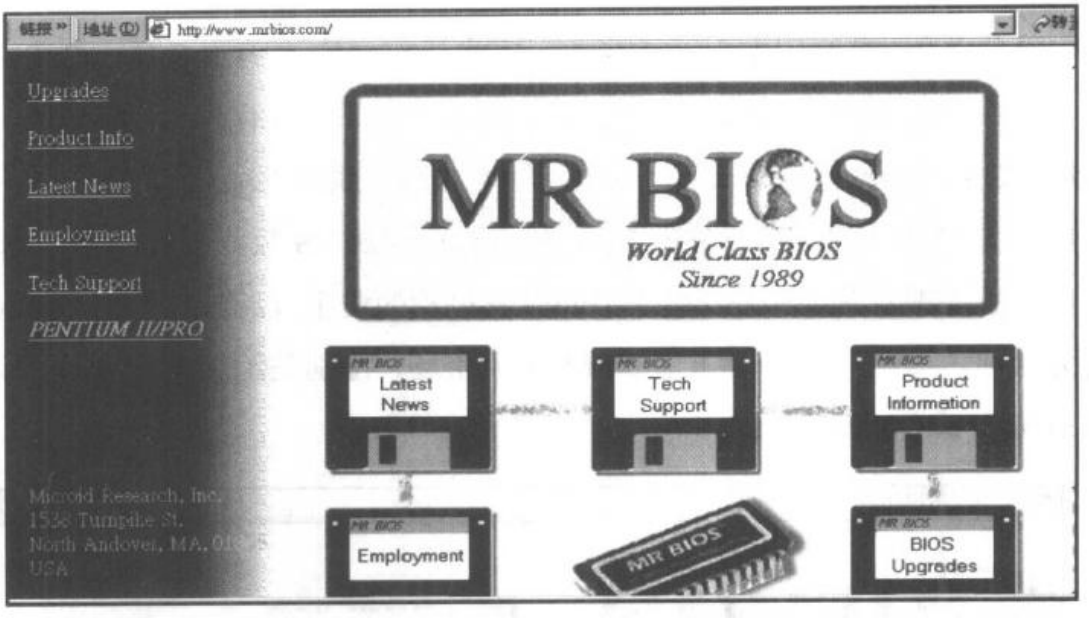


图1-13　Microid Research (MR BIOS) 的网站

以国际上计算机集团或主板厂商而言，他们掌握 BIOS 的方式大致如下：

① 完全自行开发，如 IBM、COMPAQ、Toshiba、宏碁、神通等厂商。完全自行开发的好处是可以自行掌握关键性的技术，省下购买 BIOS 授权的费用，最重要的是可以避免向合作的 BIOS 软件厂商，透露出任何正在秘密研发中的产品信息。

但一味的坚持从上到下、从内到外的自主研发，除了要自行负担 BIOS 的研发人力之外，面对今日如此竞争激烈的信息产业环境，就连那些大公司 IBM、COMPAQ、Toshiba 也都开放部分器件，甚至全部计算机都由国内 OEM 代加工生产，如何以经济的成本，实时开发新规格的产品，才是进军国际市场的策略。

② 由专业 BIOS 厂商开发并买断，自行依产品规格做中等幅度的改写，例如华硕、微星、技嘉等一线主板或系统厂商，多半是采用这种方式。由 BIOS 厂商提供源代码给他们，提供一些基本的 BIOS Function Call，以及计算机平台的基本规格实例(如 PnP，APM，DMI，USB，APM/ACPI)等等，除了这些标准规格的支持以外，BIOS 厂商并不负责修改与技术支持，一切由上述厂商的 BIOS 工程师自行处理。

接下页

当系统厂商取得源代码之后，他们再根据自己产品的需要来作修改、补充。像修改设置画面、画面中文字，甚至添加一些具有创意的功能，如CPU 频率、电压的自动检测与线上调整等。著名的有华硕的 JumperFree 手动自排调整、技嘉的 DualBIOS 技术，以及微星的 LiveBIOS 线上更新等功能。

③ 从专业 BIOS 厂商中取得授权，并请 BIOS 厂商做一些必要的技术性协助。这些厂商除了取得源代码外，还要求 BIOS 厂商协助修改 BIOS 的部分源代码(如厂商商标)，加入特别的功能、规格等(例如加入防毒软件、各种语言切换等)；有时候还要针对厂商的主板特别器材、零部件去“特别调配”，这部分当然又要特别加钱啦!

一般 BIOS 的授权费用，是由系统厂商与 BIOS 厂商签订一个套数之后，由 BIOS 厂商提供对等套数的雷射防伪贴纸，一张从大约一、二美元到五美元不等，根据采购总数的多少而定。而源代码或翻译后的机器码，由系统厂商自行烧到主板上的芯片，从BIOS所取得的源代码，则依厂商设计的计算机平台规格与特性需要做必要的修改，编译连结成功后可在这个特定计算机平台上执行的 BIOS 监控程序(这个动作称为 Porting 移植)，再由厂商自行烧写到主板上的芯片中；BIOS 监控程序烧写后，贴上防伪标志，一套具备合法BIOS授权且可开机运行的主板(或计算机系统)就可以正式出厂了!

BIOS 工程师通常要熟悉汇编语言 (Assembly) 、C 语言等，而且要仔细阅读与各种硬件设备有关的规格定义与技术文件，如各种 CPU 的 BIOS Programming Guide，与操作系统相关的规格，如 Win95 的 PnP (Plug and Play)，ESCD (Extended System Configuration Database)，ACPI (Advanced Configuration and Power Interface) 等。总之，要时常去了解最新的各种设备规格，了解最新操作系统、应用程序与硬件规格的动态，这些都是 BIOS 程序设计师经常要做的功课。

对于硬件部分，也要有一定程度的了解，要知道各种零组件的 Data Sheet，是怎么初始化 (Initializing)、调整，从显示字符、接受使用者按键信息，到控制 CPU、芯片组、内存、软盘、硬盘等所有琐碎的细节，都要自己来一步一步地去控制；但万一 CPU 或芯片组出现错误，或者跟某一种类型的芯片发生不兼容的问题，这时得根据问题的严重性来进行某种程度上的搭配与解决。

还有 Flash 内存芯片的特性是，容量越大价钱越昂贵，而且现在缺货，因此主板厂商在做主板零件采购的成本分析时，当然希望采用的 Flash 芯片容量越小、越便宜越好；而 BIOS 程序代码长度能够压缩得越小，就越能烧到容量较小的 FlashROM 芯片，以缩减主板的成本，因此大多数 BIOS 内部的程序模块，都是以预先被压缩 (Compressed) 的形式存储的，有必要时才被解压到映射内存 (Shadow RAM) 来执行，丝毫都浪费不得。

总之，开发、维护 BIOS 是一件技术含量很高的工作，而这方面的专家又相当稀少，但也因为如此，各主板厂商愿意用“高薪”加“高配股”的方式，来留住 BIOS 监控程序研发人员。当今一流主板厂商的 BIOS 研发人员，光是薪水加上配股，年薪往往是以七位数字来计算的！

况且会写 BIOS 的专家，对汇编语言具有一级的超凡实力，对硬件设备的规格掌握也了然于胸，将来即使转移到显卡、磁盘控制卡或其他硬件去开发 ROM 程序，想必也是游刃有余的；甚至对于今后 PC 时代便携式信息获取设备 (Information Appliance,IA) 的研发，也绝对不成问题。总之，这类人才进军 IA 市场，前途绝对是一片光明的。

1-4 简介美国安迈(AMI, American Megatrends Inc.) 公司

由于这本书由美国安迈 (AMI,American Megatrends Inc.) 公司提供，并开放完整的源代码以及相关的技术支持，这可谓是跨入 21 世纪的创举！因此，且容笔者花一些小小的篇幅，为这家走向源代码开放 (OpenSource) 潮流的一级 BIOS 软件厂商，做较为详细的介绍。

AMI 于 1985 年，由 S. Shankar 与一些满怀理想的工程师们创立的，总部设于美国乔治亚州首府亚特兰大市 (Atlanta)，在欧洲如德国慕尼黑 (Munich)、英国，以及亚洲如南韩汉城、日本东京、印度等都设有分公司或办事处，主要业务除了 BIOS 研发之外，也负责 USB 硬件设备ROM、RAID 控制卡ROM 等相关技术的支持与服务；而 AMI 也有开发服务器等级的主板，以及最近的 Megathlon (AMD Athlon主板) 纯硬件系统的成品。

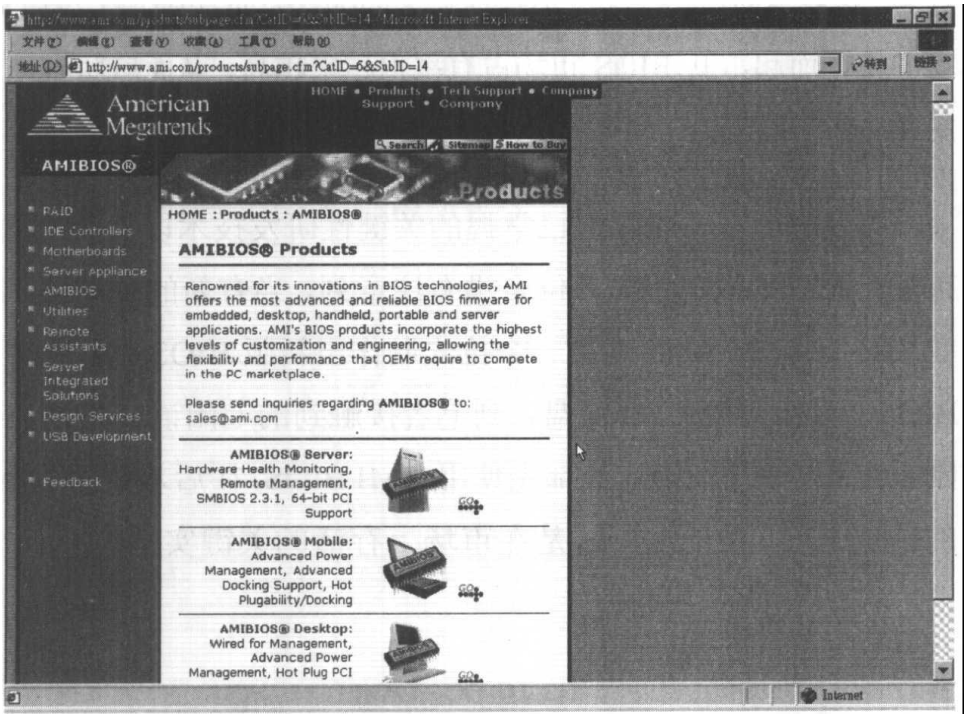


图1-14　美国安迈 (AMI,American Megatrends) 公司

在笔者与 AMI 的合作过程中，处处可以看到该公司卧虎藏龙，个个汇编语言功力了得。AMI 的创办者之一是一位印度人，本身的软件编写功力，以及硬件线路设计能力十分了得，据闻 AMI 曾经参与 Intel 486 处理器“内部微程序代码 MicroCode 的设计”。在不到十年之前的 1991 年，大约是 386、486 处理器的年代，AMI BIOS 与 AMI Keyboard BIOS (AMI 自行研发的 8042 键盘控制器) 是攻占台式计算机、主板的两大软硬件的关键器件，而当时 AMI BIOS 也以较细致的选项与菜单化、控制条的界面设计，成功地击退了单调接口的 Phoenix BIOS，而夺得 BIOS 市场的王位。

到了 1995 年，全球 50% 以上的个人计算机，内部采用的是 AMI BIOS，虽然竞争强敌 AWARD (惟尔科技)，在 Pentium (586) 时代，美国环保计算机规范出现之际，借着较低廉的授权费用，先行攻下了华硕等一级厂商的 BIOS 订单，趁着 AMI 产品青黄不接时趁势而起，因而在强调 Cost Down 的主板市场中飞崛而起，其 BIOS 市场占有率也越来越高，从而威胁到 AMI BIOS 的霸主地位。

即使如此，AMI 以自己掌握的关键性研发技术以及合理的利润，仍然在市场上屹立不动。不知各位有没有注意到，当有新的处理器、新芯片或新产业规格出现时，AMI 是第一个抢先制作出新版 BIOS 的厂商。从最早的 Win BIOS、直接支持 USB 键盘，到笔者接触到的七国语言切换，甚至在各主板刚推出新产品时，第一批都先使用 AMI BIOS，往后才使用到 AWARD BIOS 的情况来看，“技术”才是在市场上存活的关键实力。

AMI 至今仍供给 55% 的 OEM 厂商，在各种硬件设备BIOS、RAID 控制卡等 ROM 技术上提供支持与协助。主要的 OEM 客户厂商像美国惠普 HP (Hewlett-Packard) 、戴尔计算机 (Dell Computer)、Gateway、NEC、Unisys 等，至于另外一些主板厂商，如技嘉、微星、鑫明 / 精英、浩鑫,以及笔记本电脑的厂商如华宇、致福等，也是目前 AMI 的主要客户群之一。事实上还有许多签订保密合约 (Non-Disclosure Agreemnet,NDA) 的客户，由于不愿意曝光，就不在这里多说了。

在信息界如此竞争激烈的时代，一时占有率的高低不能够代表一切。在 1999 年 AMD 连续第三年亏损时，业界都没想到 AMD 会凭着 Athlon (K7) 处理器的先进结构与优异性能，硬是打出一条坦途，成为 2000 年对龙头 Intel 最具威胁性的新型处理器霸主。而以 AWARD BIOS 在台式计算机 (特别是主板市场) 占有率如此之高，仍然被 Phoenix 凤凰科技所并购，并且继续以 AWARD 的名义行销其台式计算机 BIOS (笔记本电脑 BIOS 仍以 Phoenix 为主)。如果低价、低授权金额的政策奏效，今天应该是由台式机市场称王的 AWARD 去并购 Phoenix (毕竟台式计算机市场比较大)，而不是由笔记本电脑 BIOS 市场之王 Phoenix 去并购，您说是不是?

AMI 有完全自主掌握的源代码关键技术，具有直追 Linus Torvalds 开放源代码的开阔胸襟，笔者能够与该公司充分合作，合力完成本书，为ROM BIOS 的人才培训作一点贡献。相信在本书以及 AMI BIO 源代码开放的第一手资料的帮助下，有程序基础的读者能够一窥十多年来主宰计算机最深层灵魂中的秘密，对 AMI BIOS 的规划与程序代码的开发能够有全盘的了解；即使日后接触到其他 BIOS 的开发工作，想必也能够游刃有余。

据说 AWARD 母公司 Phoenix 即将上调 BIOS 的授权标签费用，对“成本”向来重视的主板业主们，从成本上考虑，极有可能见风转舵，重回 AMI BIOS 的怀抱。再加上 AMI 工具程序齐全，不但画面亮丽、操作容易就手，功能也十分强大。所谓十年河东、十年河西，AMI BIOS 在大环境又对该公司有利的情况下，可预期 AMI 在 BIOS 市场占有率将会呈现跳跃性的成长，朝向 BIOS 市场的王者之位。

最后，简介一下 AMI BIOS 产品线，它大致可分为下列各项产品：

① AMIBIOS (R) Server 针对高效能服务器所开发的ROM　BIOS 系统

② AMIBIOS (R) Mobile 针对笔记本电脑所开发的ROM　BIOS系统

③ AMIBIOS (R) Desktop 针对一般台式计算机所开发的ROM　BIOS系统

本书主要以 AMIBIOS Desktop，也就是台式计算机的 ROM　BIOS系统为讨论的对象。在本章节的最后，我们列举出其主要的功能与支持的各种接口规格 / 协议：

Key Features & Functionality (主要特点与功能)

主要特点	功　　能
APM 1.2 & ACPI (S1/S2/S3/S4 & D2/D3)	APM/ACPI接口规格的支持与实践
SMBIOS 2.3.1 Compliant	SMBIOS 2.3.1 版
PC 98 & 99 Compliant Quick Boot Support	符合最新 PC 98 & PC99 制定规范 快速开机功能
FlexbootSupport	弹性 / 多样化外部设备开机功能
Legacy & USB Device Boot	传统 USB接口的软盘 / 硬盘 / 光盘开机、I2O 设备开机与网卡开机等功能
Silent Boot Support with OEM's Logo	支持超大 OEM 商标画面显示
Legacy USB Support	USB设备在传统环境 (DOS) 下的直接支持 / 驱动能力
USB 1.1 & P1394a Fireware (Universal HCI & Open HCI)	USB 1.1 与 P1394a 高速总线支持
WfM (Wired for Management) 2.0 PCI 2.1 & Hot Plug PCI	WfM 2.0 远程管理功能 PCI 2.1 与热插拔 PCI 支持能力
64bit PCI-X	64 位 PCI-X 规格支持
Resume on Alarm, Modem Ring and LAN	定时开机、调制解调器来电开机与网络遥控开机功
Legacy Free Ready	充分准备“非传统结构”计算机 (无 ISA/uperlO...) 平台的实际支持
Multi Language Support	多国语言支持,包含单字符语系 (欧洲) 与亚洲双字符语系,支持繁体、简体中文、日文平假 / 片假 / 汉字、韩国字等

CPU/Chipset (处理器与芯片组支持)

- 支持所有目前x86结构(Intel，AMD，VIA/Cyrix)的处理器，以及 Transmeta Crusoe 处理器
- 支持 Intel Itanium (IA-64) 结构的处理器
- 支持目前市面上所有芯片组，含 Intel，VIA，SiS，ALi，Micron，OPTi 等芯片厂商
- 支持所有 Super I/O 以及音频芯片
- 支持 MPS (MultiProcessor Spec) v1.1，v1.4 ，以及 v1.5 协议。

完整的工具程序

AMIFLASH.EXE	可在 Windows 环境下直接执行的 FlashROM 烧写程序，支持各种1Mb/2Mb/4Mb 容量的各种品牌 的Flash 内存
AMIBCP.EXE	窗口化 / 菜单驱动 (Windowed/Menu Driven) 工具程序，提供模块化添增 / 删减、修改默认参数值与寄存器数值以及供 OEM/主板厂商的画面选项设计程序、OEM 商标的修改程序、新 CPU 模块添增程序、取图程序等。这部分将于第9章再做详细介绍。

最后，跟据AMI 的产品消息，AMI 即将针对 NS570 微控制器开发所谓的 EC BIOS 产品。EC 是 Embedded Controller (嵌入式控制器) 的缩写，结构上可以被视为 8042Keyboard Controller(KBC) 之后的微控制器的延伸，EC 目前普遍应用在具备智能型节电管理功能的笔记本电脑设计中，它肩负着键盘、鼠标、电池电源以及温控检测的任务。

AMI EC BIOS 能够搭配 NS570 这个微控制器，其监控程序可以存储到 NS570 控制器内部的 MicroROM，或者跟主板的 BIOS 监控程序一同存储于 FlashEPROM 中共享，但在译码结构上分隔各自处理的模式。AMI 的EC BIOS 可以提供笔记本电脑内部 的Matrix 矩阵译码型键盘、外接传统型键盘的仿真作业，以及多达 10 组的一般热键 (General HotKeys) 、12 组功能热键 (Function Hotkeys,Fn) 的重新定义，以及最多 3 组 PS/2 键盘或鼠标设备的支持能力，支持先进的 APM/ACPI EC 接口模式，当笔记本电脑进入待机甚至关机模式时，肩负着外部设备电力调度及智能型电池的电力侦测 / 回报等任务。

通过 EC BIOS 与 NS570 EC 的合作，可以使笔记本电脑厂商发挥出创意与设计弹性，在温度监控与电源管理上具有更加的自动化，尤其对强调智能型电源管理的笔记本电脑而言，是一项不可多得的技术解决方案。

在本章节的最后，我们要介绍 AMI 公司对新进人员的培训课程：

周	课程大纲	辅助课程内容摘要
第	公司简介	Assembly 基本语法测验
	部门简介	Make，Nmake 文件维护
一	BIOS 简介	Debug 程序除错
	Chipset 简介	简介各家芯片组规格与特性差异
周	PCI CONFIG SPACE	检测 PCI 设备的程序操作 (RU、IOU)
第	SOURCE Sequential List	BIOS 源代码目录结构解说与源代码浏览
	Hardware Data Sheet	硬件技术资料解读
二	Make BIOS	实际编译制造一组 BIOS 程序文件
	BIOS DATA BIOS	技术资料与相关结构
周	SHEET/Structure	BIOS 工作区分与任务分配
	BIOS Work 简介	
第	START Project	小型简易计划的分配与线上实作
	(进行计划分配)	
三	Memory Sizing/SPD	内存容量 / 型号检测的实作
	POST ADJUST	系统资源 (I/O、Memory、CMOS) 读写实作
周	GPC INIT/ACTIVE	研读 GPC 模块与笔记本计算机平台
	USB	USB 控制原理与实作
第	Supe r I/O CHIPSET	关于 SuperI/O 与其他芯片的实践
	SMB IOS	读取 SDRAM的SPD 参数与 ClockGen 设定 四
四	ACPI、GPC (S3、S4)	电源管理的实践、ACPI、ASL 语言设计
	DMI	GPNV 存储器的读写实作
周	FLAS HPART	Flash Programming 快速存取存储器

一般而言，一位具备基本汇编语言能力的新员工，大约经过四个星期的密集课程培训之后，就可以分派工作，由 Team Leader 分派某个项目 (Project)。一般例行性的 porting 任务应该不成问题，但是当涉及到新型 CPU 、芯片组或硬件变动、错误时，则由 Team Leader 协调，召集不同部门的资深人员一同协商解决。

CHAPTER

2

硬件基本概论与规格导读

为了有助于本书后续章节的进行，本章将对从 1996 年以来所出现的各种 CPU 、芯片组与主板平台，所牵扯到的专业术语、技术规格做一个概略性的介绍，如果读者对这些术语已经非常了解，也可以快速翻阅或直接跳过本章，直接进入下一章的阅读。

如果读者对这些硬件规格、外部零组件需要更进一步的了解，笔者建议您不妨参阅市面上相关的硬件 DIY 书籍 (如笔者一系列拙著 97、98DIY 书籍) 或其他 DIY 杂志。

2-1 计算机处理器、主板、芯片组的介绍

CPU(Central Process Unit)中央处理器,微处理器(Processor)

CPU 是整个计算机系统的重要运算核心。在传统教科书对 CPU 的定义中：CPU 由控制单元 (CU) 、算术与逻辑运算单元 (ALU) 与内存单元 (MU) 所组成。早期中大型计算机的 CPU，其实是以设计复杂的电路板形式呈现，随着半导体科技的进步，厂商将 CPU 电路以芯片的形式制造出来，这种 CPU 芯片又称微处理机或处理器 (MicroProcessor 或 Processor) 。

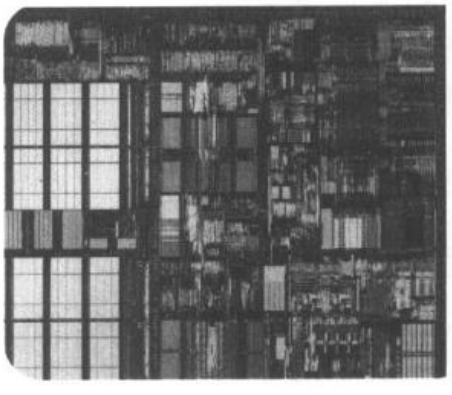


图 2-1　CPU 是计算机最重要的运算核心零组件 (此为 Intel Pentium III集成电路)

近二十年所发展的计算机系统，几乎都是采用处理器芯片作为计算机的CPU，由于大家对CPU这样的称呼已经习惯了，所以当您在书上、媒体或技术文献上，看到无论是CPU、中央处理器或处理器(Processor)，这些术语指的都是同一个东西。

计算机结构的差异，主要是取决于CPU设计上的差异。例如我们个人计算机所使用的处理器，通称为x86构造——就是从以往8086/8088，80286，80386，80486到Pentium，Pentium II与Pentium III等处理器，所沿袭下来的指令/数据格式，以及内存&I/O寻址模式等的通称。

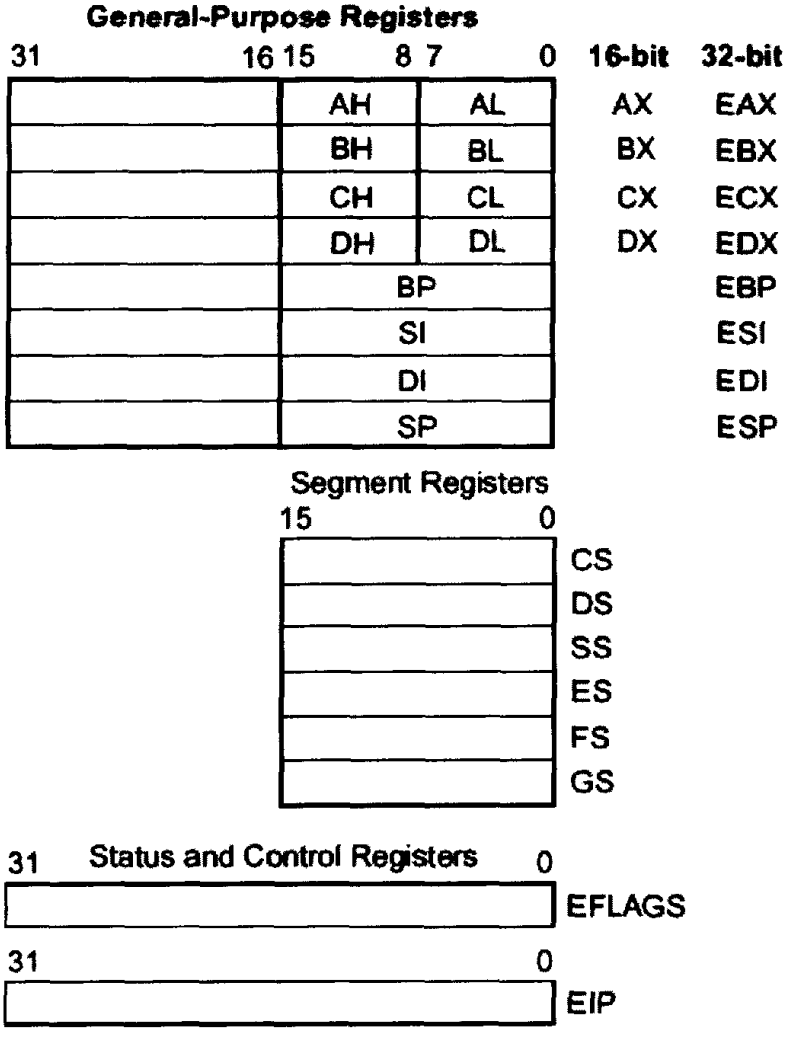


图2-2 x86构造的8/16/32位寄存器格式

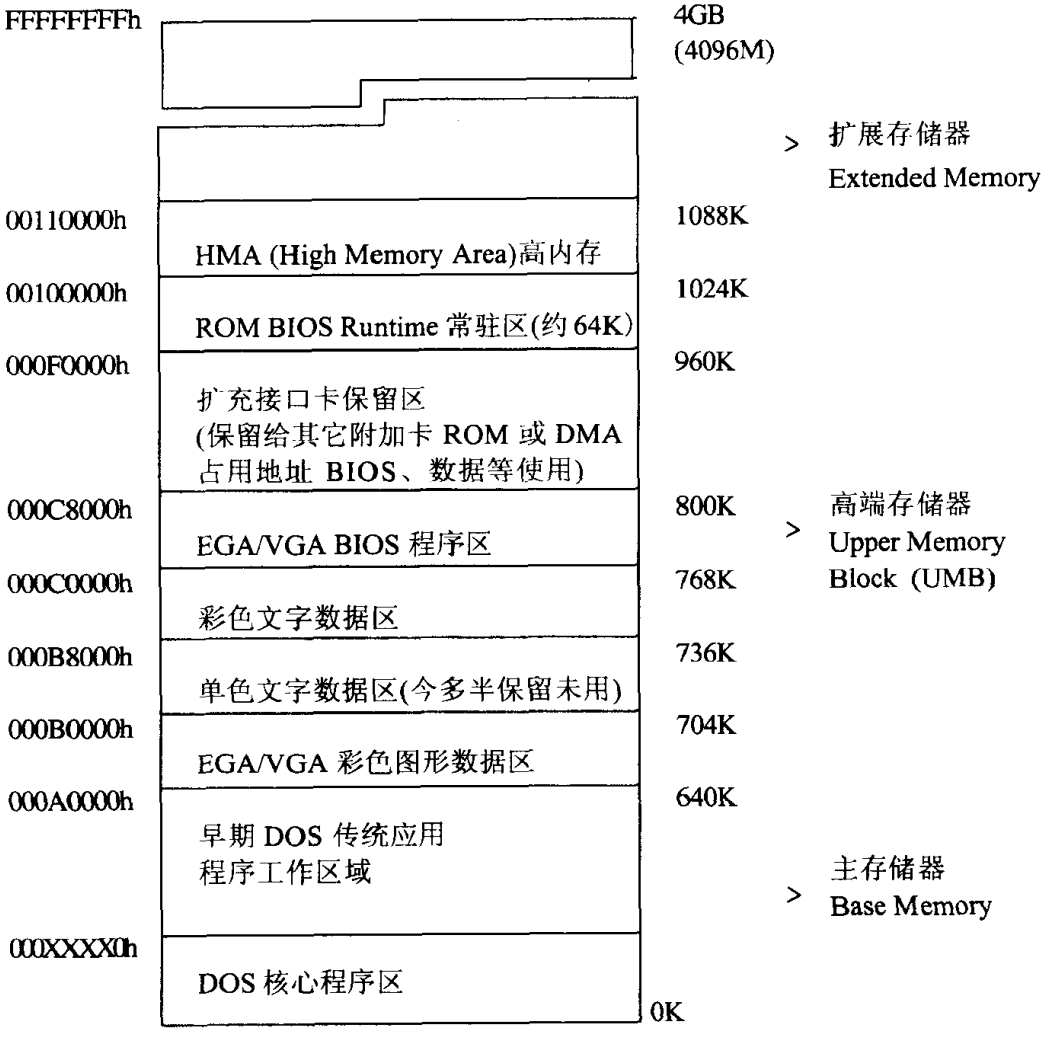


图 2-3　x86 构造的内存规划分布图 (386 以上)

至于 AMD，VIA/Cyrix 设计的处理器，走跟 x86 结构兼容的设计，意思就是它在设计上保留了跟 Intel 处理器所有 (或绝大多数) 指令集、数据格式在内存与 I/O 空间的划分上，保持着相同或近似的设置，除了系统类等级的软件 (如 BIOS，OS 或部分驱动程序等) 需要做小部分修改之外，绝大多数针对 Intel 处理器开发的软件，也可以在 AMD，VIA/Cyrix 或其他兼容 x86CPU 上执行，因此这类兼容型 x86 处理器，我们也把它们列为 x86 结构的一族。

以下是我们整理最近五年所出炉的 x86 结构的处理器品牌、型号与相关资料型号：

处理器品牌	接口构造	处理器名称/型号（代号，制程）	发表年代
Intel	Socket 7	Pentium、Pentium MMX	1995～1998
	SLOT 1	Pentium II (Klamath, 0.35 μ) Pentium II (Deschutes,0.25 μ) Celeron (Covington Mendocino)	1997Q2 1998Q1 1998～2000
	SLOT 1	Pentium III (Katmai,0.25 μ) Pentium III (Coppermine,0.18 μ)	1999Q1 1999Q4
	SLOT 2	Pentium II-XEON Pentium III-XEON	1998Q3、 1999Q3
	Socket370	Celeron (Mendocino)	1999Q1～ 2000Q1
	FC-PGA370	Pentium III (Coppermine-256) Celeron (Coppermine-128)	1999Q4 2000Q1
	Socket423	Pentium4 (Willamette)	2000Q4或 2001Q1
	Socket478	Pentium4 (Northwood)	2001Q1或Q2
	整合型	Timna	2001Q2(预定)

处理器品牌	接口结构	处理器名称/型号（代号）	发表年代
AMD	Socket 7	K6、K6-2 (100MHz FSB)	1997Q2、1998Q2
		K6- III (256KB L2 on Die)	1999Q1
	Socket 7	K6-2P、K6-2+ (128KB L2)	1998、1999Q4
	360pin CBGA	K6-3P、K6-2+、K6- III	2000Q1
		+(256KB L2)	
	SLOT A	Athlon (K7)、	1999Q2、1999Q4
		Athlon(K75,0.18 μ)	2000Q2
		Athlon (Thunderbird,0.18 μ)	
	Socket A	Athlon (Thunderbird,0.18 μ)	2000Q2
		Duron (64KB L2 On-Die,0.18 μ)	2000Q2
VIA	Socket 7	Cyrix II (M II)	1998Q2～1999Q4
	Socket 370	Cyrix III(Joshua)、	2000Q1、2000Q2
		Cyrix III(Samuel)	
	Socket 370	Cyrix IV (Samuel2,64KB L2	2001Q1 或Q1
		on Die)	
IDT Cenatur	Socket 7	Winchip C6、	1997Q2
		Winchip2、Winchip2-3D	1998Q2、1998Q3
		Winchip3	1998Q4～1999Q1
Rise	Socket 7	mP6、	1998Q4、1999Q2
		mP6- II	

对于BIOS 程序设计者而言，新一代处理器为了兼容性 (Compatibility)，多半会保留前一代处理器的指令(除非该指令有严重错误或者极为少用而被废除)，因此以往所编写的 BIOS 程序代码，只要能以类似 CPUID 的指令，来检测新 CPU 硬件特性，并对新的寄存器做填值或初始化 (initializing) 的动作，就可以在这块 CPU 顺利地执行，而其余大部分程序代码是不需要修改的。在后面我们会再深入讨论，当新的 CPU 、芯片组或主板平台出现时，BIOS 是最常修改的部分。

主板(MainBoard 或 MotherBoard,简称 M/B)

主板是负责处理器、内存、显卡与外部接口的联系的电路板，上面焊有必要的芯片与电阻、电容等相关组件，只要 CPU 、内存、显卡等装置通过连接器 (Connector) 跟主板连接，接上电源之后，就构成了最小的可运行的计算机系统。

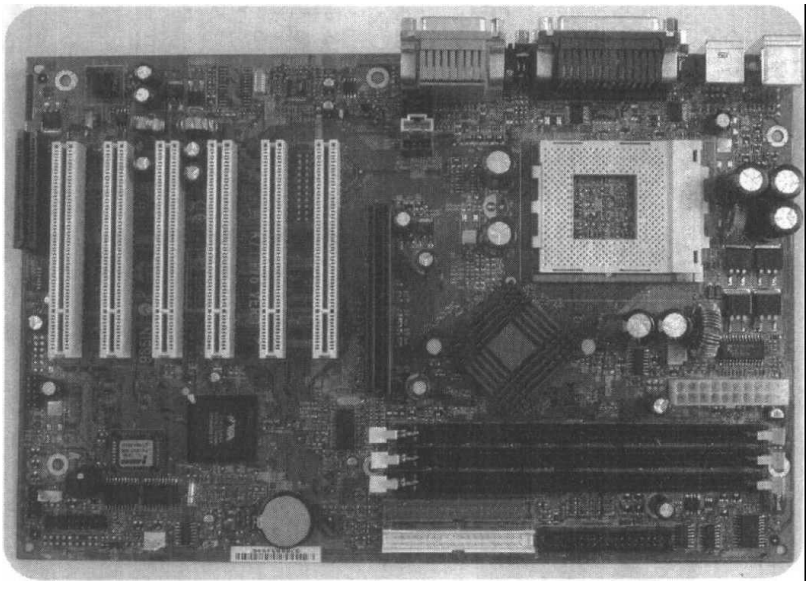
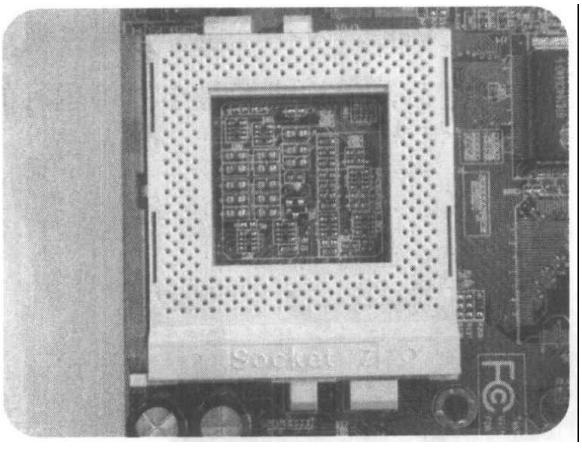


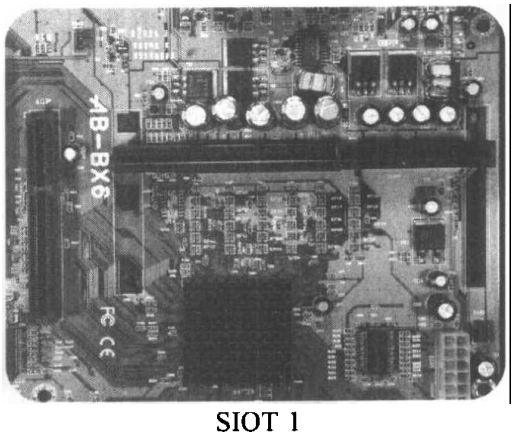
图 2-4　主板

连接器 (Connector) 是负责将两种装置嵌合、让电路流通的中介装置。连接器其种类繁多，像主板供 CPU 、插槽 (SLOT) 或插座 (Socket)、适配卡安插的扩充槽等等，其实广义的来说，都算是众多连接器的一种。

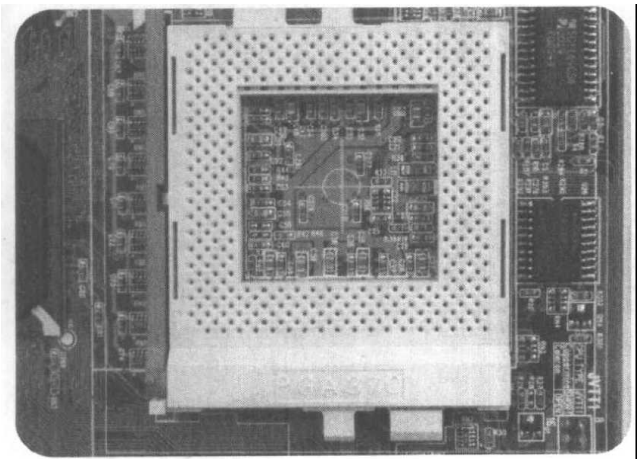
由于每一代的 CPU ，都会因为对总线讯号定义、封装方式的差别，而重新设计不同的插槽或插座来与主板其他的组件连接，厂商也以搭配某款 CPU 的连接器(插槽 / 插座)形式来区分他们的产品类别。像用 Socket370 结构，其实指的就是用 P6 Bus 设计的 Intel 主板；走 Socket462 (Socket A) 结构的，其实指的就是用 AMD EV6 Bus 设计的 AMD 系主板。



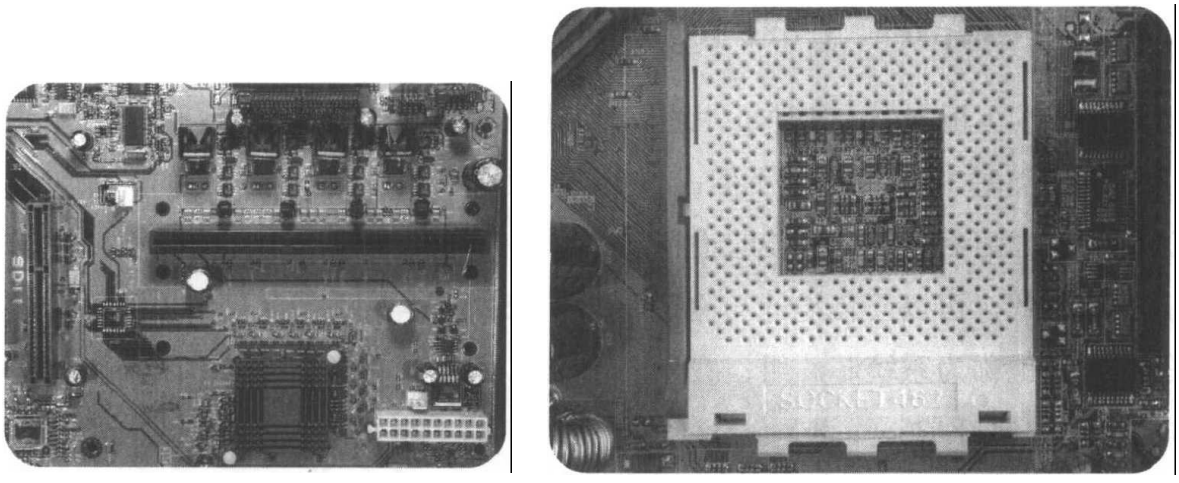
Socket 7



SIOT 1



Socket 370



SLOTA　　Socket462 (Socket A)

图 2-5　各种类型的连接器形式

近代主板的结构、功能与规模，绝大部分由 CPU 决定，其次是由芯片组所决定，因此对于 BIOS 厂商而言，跨世纪的 BIOS 移植技术难度比较高，相对的同一代处理器的 BIOS 比较容易被移植 (Porting) 。例如针对 Pentium 主板开发的 BIOS，如果要移植到 Pentium II，可能要花数个月时间来全面改写；但同样是从 Pentium II 系统转移到另一套 Pentium II 系统，也许就只要花几星期的时间就可以编写完成并成功地移植过来。

而同一代又同一家 (甚至同一种) 芯片组设计的主板平台，其 BIOS 移植的技术难度就比较低了。例如 BIOS 厂商已跟某 A 大厂合作，开发好了针对 Pentium II / III处理器，而且主板是采用 Intel 440BX 芯片组设计的，那么当另一家 B 厂商也有类似平台 (也是 440BX 芯片组) 的开发需要时，因为是同一家的芯片组结构上又更接近，通常 BIOS 厂商只要经过短时间的修改，就可以移植到 B 厂商的主板，接下来的创意与维护任务，就交给 B 厂的 R&D 工程师接手了。

芯片组 (Chipset、Core logic)

早先的主板上面，各种组件之间的连接与协调电路极为复杂，动辄使用到上百块各类型 IC、电容，不但占用 PCB 板面积，也造成除错上的麻烦与无谓的成本。随着半导体科技的进步，主板上的许多线路，已使用几个大型 IC 芯片来取代，可以简化设计与除错时间、减少成本，这些专攻主板简化设计的芯片，就称为系统芯片组；在电路学上也有人以“核心逻辑 (Core Logic) 电路”来称呼它。

主板芯片组在经过长期的整合，从 Intel 在 97 年 Pentium MMX 时期推出编号 82430TX (俗称 430TX) 芯片组开始，确立“北桥芯片 (North Bridge)”与“南桥芯片 (South Bridge)”南北共治，处理器一同统治主板江山的局面。前者负责内存与 AGP 接口的显卡、内存与 CPU 处理器之间的沟通，后者则负责 PCI 外围、软盘、硬盘、USB 装置、鼠标、键盘等设备。

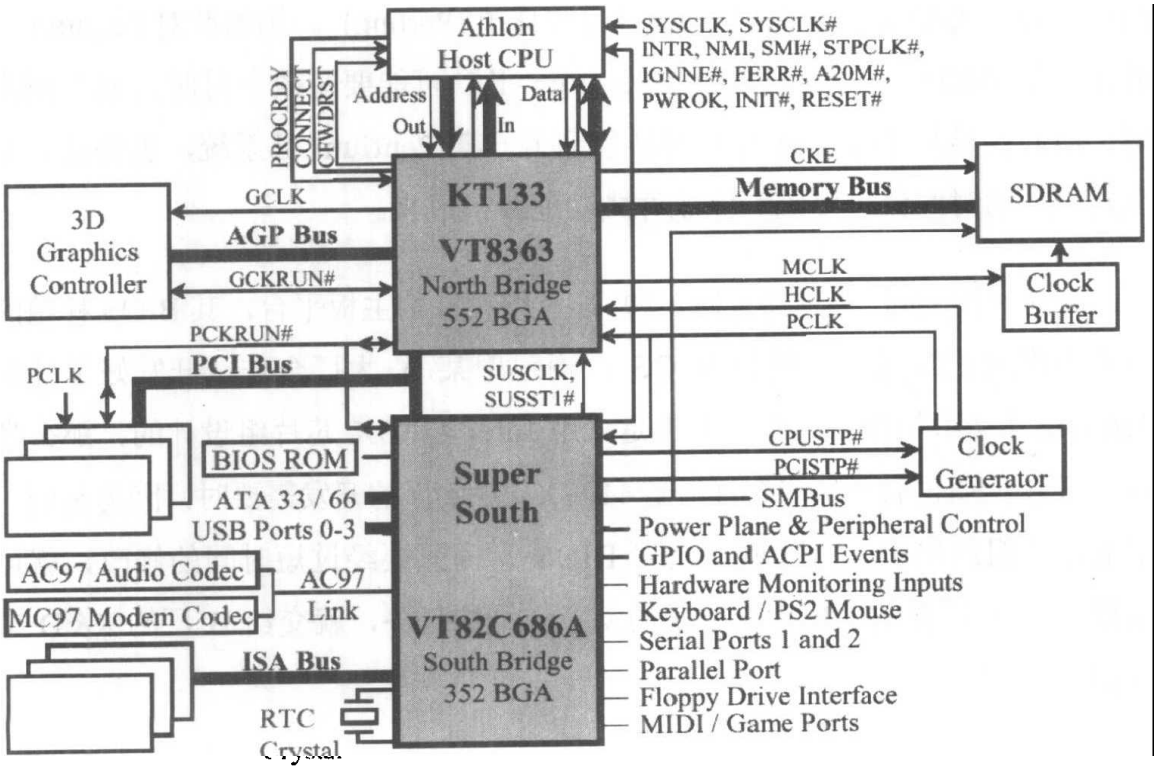


图 2-6　主板系统平台与芯片组构成图 (以威盛 Apollo KT133 为例)

当然也有业内人士把北桥、南桥芯片的结构再整合成一块芯片，像 SiS 的 540、630 以及刚发表的 730s、630s 等芯片，但是从 BIOS 的观点，以及系统软件、硬件规划的角度上，我们还是认为这种芯片具备北桥——南桥结构，只不过在实质的外观上，两种电路单元已融入一块芯片内部，并且各自独立运作着。

如果我们把一套计算机系统用方块图来比喻成一个国家，CPU 相当于总统，北桥芯片则是中央政府，南桥芯片则是地方政府，那 BIOS 又是什么呢？当然就是维系整个国家政经体制的宪法或基本法令，有了 BIOS 监控程序的运作与协调，北桥／南桥芯片才能相辅相成，与 CPU 、适配卡等良好互动。

独立型芯片组与整合型芯片组 (Integrated Chipset)

因应低价计算机的兴起，以及计算机体积微缩的趋势，芯片组厂商致力于将许多新兴功能／电路，直接整合进传统的北桥／南桥芯片组内，使得计算机系统成本压低、体积微缩之际，仍然可以具备当今多媒体／互联网计算机的性能与规格上的要求。在这种低价、高功能的市场趋势下，造就了整合型芯片组与整合型主板的兴起。

所谓整合型芯片组，以 1999 年到 2000 年的定义而言，传统的北桥芯片整合了 2D/3D 绘图显示芯片的功能，另外南桥芯片也整合了 Super I/O，AC97 等线路或规格，以整合型芯片所设计的主板，大体上只需要两组最重要的北桥／南桥芯片，就可以直接连通 CPU 、内存、软硬盘、USB 等装置。而整合显示／音效功能的芯片组设计的主板，只要将 CPU，RAM 安装好，接好硬盘、软盘、键盘、鼠标等设备后，接上 Power 一打开电源，就形成一个可运作的计算机系统，安装上简单方便，同时也节省整个系统的购置成本。

这种整合显示功能的芯片组，以威盛 (VIA) 的 Apollo MVP4，SiS (硅统) 的 530(如图 2-7 所示)，540 (Socket7)，620，630 (SLOT1/Socket370)，以及 Intel 810/815e 芯片组最为著名。

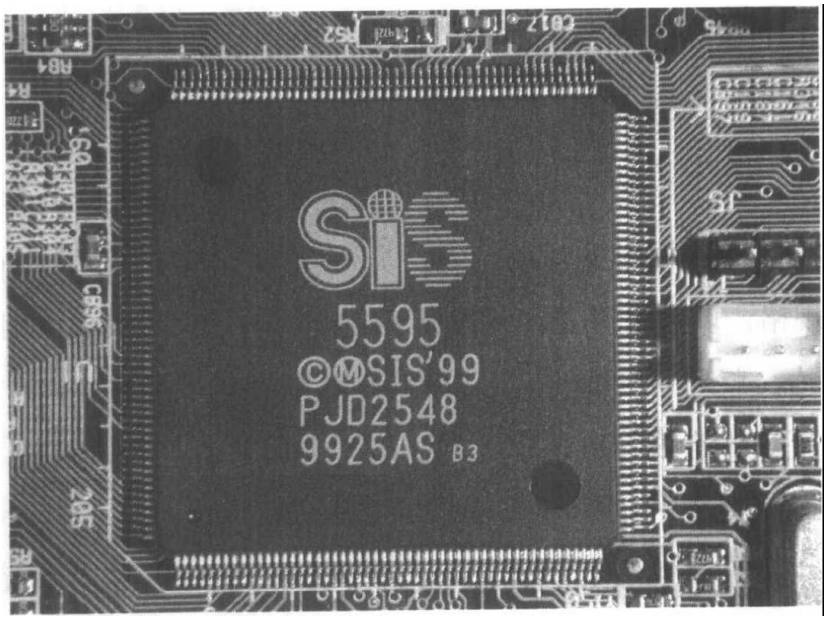


图 2-7　此为本书要介绍 / 实作的主板芯片组——SiS530

至于没有整合显示电路的芯片组 (北桥芯片)，由于需要外接 AGP 或 PCI 接口的显示卡，计算机才能显示，因此这种芯片组就以独立型芯片组 (或传统芯片组) 称呼，它们的重点在于提供消费者可以扩接更高效能的 AGP 接口显示卡，以提升计算机的图形运算 / 处理能力，走的是属于比较高档、讲究效能与扩充性的玩家市场。

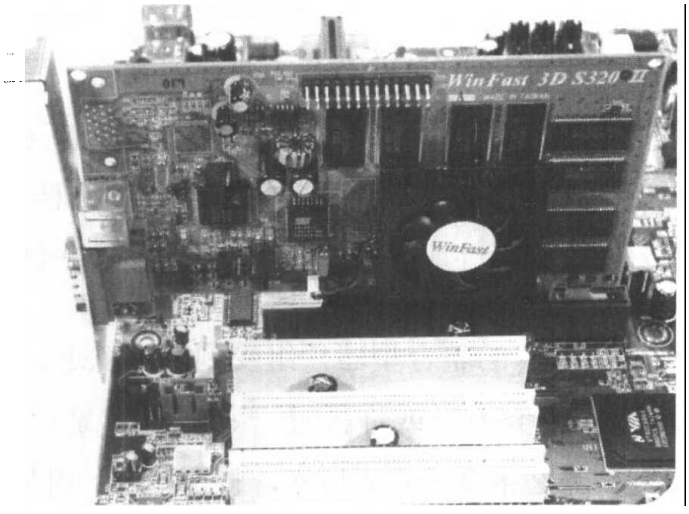


图 2-8　传统型芯片组须外接 AGP 接口高阶显示卡，以效能 / 扩充性取胜

还有一种被称为 SoC (System On Chip) 整合单芯片，它除了整合传统的北桥芯片、显示电路之外，甚至连 x86 结构的处理器都整合进去，整套计算机几乎只要一块芯片 (或搭配少数 IC 芯片) 就可以搞定，最著名的有美商国家半导体 (NS — National Semiconductor) 的 Geode 处理器，其前身来自于 Cyrix MediaGX 处理器，而在 Intel 也要开发所谓的 Timna 处理器，也相当于在 810 芯片组中的 GMCH (Graphic Memory Control Hub) 再加入 Celeron 处理器的核心运算电路，只要搭一组 ICH (I/O Control Hub，Intel 对南桥芯片的称呼，就可以构成一个完整可运作的计算机系统。

其实现今主板芯片组，或多或少都“整合”了许多以往要借助外部芯片才能享有的功能。众所周知的如 CMOS/RTC (RealTimeClock)，Super I/O (整合进南桥 South Bridge)、最近比较新颖的是 AC97 (软件音效 / 调制解调器传输) 等。如果要认真探究传统型芯片组与整合型芯片组之间的差异，似乎只在于“有没有整合图形 / 显示电路、能不能拉出一根 AGP 来加装显卡”而已。

而且在本书完稿之前，Intel 最新的 815/815e 芯片组，以及硅统科技的 630s (P6)，730s (K7) 芯片组，虽然芯片本身已经整合了图形显示电路的功能，却可以像传统北桥芯片一样，对外拉出一根 AGP 扩充槽，把北桥芯片内建的图形显示电路关闭后，可以另外接 AGP 接口的高档 3D 绘图加速卡，其扩充能力又跟传统 / 独立型芯片组相当，真是很难做严格的区分。

如果这种“可扩充型”的整合型芯片组受到热烈欢迎，很有可能所有芯片组厂商都会朝这方向去努力，或许再过一、两年，显示电路内建、AGP 接口扩充能力等早就成为必备甚至基本规格，整合型芯片组与传统芯片组的定义，可能以“有无整合基本运算能力的 CPU 电路”来区分了。

当然，天下没有白吃的午餐，整合型芯片组开发较为困难，成本也比较高昂，而绘图芯片厂商为了避免被“整”的命运，必须要继续开发更高效能的绘图芯片产品，跟整合芯片组的性能做区分，其他 IC 芯片设计业者又何尝不是如此？这样的市场形势，用一句“天下大势分久必合，合久必分”来形容，是最恰当不过的了。

Socket7 芯片组 (Pentium/Pentium MMX，K6-2，K6-Ⅲ，K6-2+/K6-Ⅲ+，Cyrix MⅡ...)

Intel	VIA	SiS	ALi
430VX 430TX	VP3、MVP3	5591	Aladdin Ⅳ Aladdin Ⅴ
	MVP4	530 540	Aladdin7

SLOT 1/Socket370 (P6Bus) 芯片组 (Pentium Ⅱ/Ⅲ、Celeron、VIA Cyrix Ⅲ)

Intel	VIA	SiS	ALi
440LX	ApolloPro(691)		AladdinPro
440BX 440ZX	ApolloPro133(693a) ApolloPro133a(694x)		AladdonPro Ⅱ AladdinPro3 AladdinPro4
820(Camino, RDRAM)			
810/810e		SiS620	
815、815e	ApolloPM133(8605)	SiS630、630S	AladdinTNT2(M1631) AladdinCyberBlade(M1632)
	ApolloPX266(DDR)		AladdinPro5(M1651)

Pentium4 芯片组

Intel	VIA	SiS	ALi
Tehama (Dual RDRAM)			
Brookdale (DDR-266)	Apollo Px266	SiS645	M1661

SLOT A、Socket A 架构 AMD K7 芯片组 (AMD Athlon、Duron 处理器)

AMD	VIA	SiS	ALi
AMD750 (IronGate)	ApolloKX133(8371) ApolloKT133(8363)	SiS730s	
AMD760	ApolloKT266(8366)	SiS735	ALi MAGiK1 (M1647)
AMD760MP (2CPU)			

2-2 主板零组件、总线等特定规格介绍

Super I/O 芯片

Super I/O 芯片的历史甚久。在早期 286、386 甚至 486 的初期，当时的主板都是接一种被称为 Multi I/O (或 Multi Function) 的适配卡，它提供必要的软／硬盘接口、打印机接口、两个串行端口、打印端口以及摇杆接口，随后 IC 设计人员将这些必要的 I/O 接口整合，发展成所谓的 Super I/O 芯片，供厂商发展 Super I/O 适配卡，或直接整合在主板上，如图 2-9 所示。

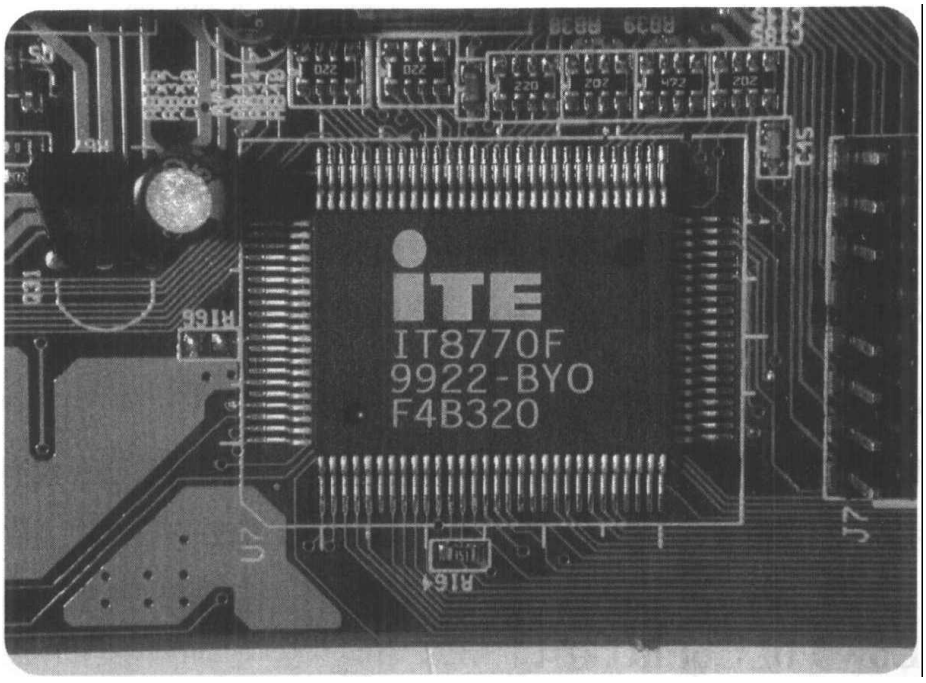


图2-9 Super I/O 芯片

Super I/O 芯片整合较为中低速率的接口，以定义而论最少都整合2S1P1G1FD，也就是具备两个串行接口 (Serial Port, COM1&COM2)、一个平行打印接口 (Parallel Port)、一个游戏 / 摇杆接口 (GAME/Joystick I/O)，还有一组软盘控制接口 (FDC Controller)。后期的 Super I/O 芯片还涵盖了红外线传输接口 (InfraRed，IR 接口)，芯片设计上可选择占用 COM1 或 COM2 来使用，如果您启用 IR 传输功能时，则必须要关掉一组 Serial Port 供红外线传输接口使用；这个切换的机制一般都是通过 BIOS 做线上设定，所以从开机时期就要决定是要使用两组传统 Serial Port,，还是保留一组使用 IR Port 做红外线传输。

研发 Super I/O 芯片的厂商，也一直不断地为 Super I/O 增添新的功能，像硬件监控功能，能够检测 CPU 温度、工作电压值以及 CPU 散热风扇的马达转速等等，有些 Super I/O 芯片还提供了 IDE 接口，可以直接连接硬盘、CDROM 等 IDE 装置。随后因应 Intel 所提出的 LPC 接口，Super I/O 芯片也从以往的 ISA Bus 走向 LPC 接口。至于主板要使用 Super I/O 的 IDE，还是使用传统南桥芯片的 IDE 来连接硬盘，这得看主板设计者的考虑而定。

其实在2000年以后所出现的芯片组，其南桥芯片多半已经将传统的SuperI/O(2S1P1G)等接口整合进去，笔者手中有几块新的主板，就已经没有 Super I/O 的设计，将来读者可能再也看不到有所谓 Super I/O 的芯片了。

LPC (Low Pin Count) 接口

LPC (Low Pin Count 低脚位数) 接口，是 Intel 于 1997 年 9 月 29 日公布的一个取代传统 ISA Bus 的一种新接口规格。并且以免费开放授权的方式，供业界所采用。

以往为了连接 ISA 扩充槽 / 适配卡、ROM BIOS 芯片，以及控制 2S1P1G 等 I/O 接口，南桥芯片必须保留一个 ISA Bus，并且连通 Super I/O 芯片，以控制传统的外围设备。传统 ISA Bus 速率大约在 7.159 (由 14.318MHz 除频) ～ 8.33 MHz (由 PCI 速率除以 4 而得)，提供的理论尖峰传输值为 16MB/s (实际不到 7MB/s)，但是 ISA Bus 与传统 PCI Bus 的电气特性、信号定义方式迥异，南桥芯片、Super I/O 芯片得多浪费针脚来做处理，主板的线路设计也显得复杂。

Intel 所定义的PC 接口，将以往 ISA Bus 的地址 / 数据分离译码，改成类似 PCI 的地址 / 数据信号线共享的译码方式，信号线数量大幅降低，工作速率由 PCI 总线速率同步驱动，虽然改良过的 LPC 接口一样维持最大传输值 16MB/s 左右，不过所需要的信号脚位数大幅降低 25～30 个，以 LPC 接口设计的 Super I/O 芯片、Flash 芯片都能享有脚位数减少、体积微缩的好处，主板的设计也可以简化，这也就是其取名为 LPC —— Low Pin Count 的原因。

	ISA	LPC
数据大小	16bit	4bit
工作频率	8.33MHz	33.3MHz*
寻址空间	2^{24} =16MB	2^{32} =4GB
最大传输频宽	16MB/s	16MB/s

注：与传统 32 位 PCI 时脉 (=33.3MHZ) 相同

当然若主板厂商因为外销市场的需求(如非洲、中国大陆或印度等)，非得加入传统 ISA 扩充槽，那就得另外借助一种符合 LPC 接口的 PCI/ISA 桥接芯片，将 LPC 接口的 33MHz、4bit 信号转成以往的 8MHz，16bit 的信号来跟传统的ISA 适配卡连通，但这种 PCI 转 ISA 的桥接芯片，无可避免地会造成主板设计面积与成本的增加，南桥芯片也得牺牲一组 PCI 槽的驱动能力，来驱动这多出来的 ISA 扩充槽与装置。

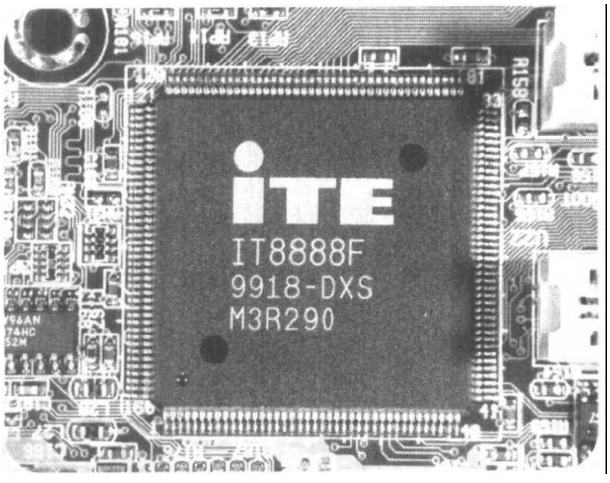


图 2-10　支持 LPC 转 ISA 的桥接芯片

在 PC99 规格中，已经建议人们废除 ISA 扩充槽与 ISA Bus，全面以 PCI 扩充槽来取代。LPC 接口的提出是第一步，随后又有 PC2001 规格中 Legacy Free PC (非传统 PC) 的提出。在最新的 PC2001 规格中，提出了所谓 Legacy Free PC (非传统 PC) 的概念，也就是一切外围设备将通过 USB 来连接、扩充，适配卡则只剩下 PCI，AGP 以及 CNR/ACR 四种规格。稍后我们会针对 Legacy Free PC 的规格再做详细的介绍。

Intel 810 芯片组以及 820，815/815e 芯片组，其南桥芯片已经废除 ISA Bus，改以 LPC 中低速率接口来跟外接 (或内建) 的 Super I/O 芯片联系，而威盛 (VIA) 、硅统 (SiS) 与扬智 (ALi) 三家芯片组厂商，也随后陆续推出类似功能的南桥芯片组。对于主板设计者而言，他们所注意的是南桥芯片所连接的 Super I/O、ROM BIOS 这部分是遵循 LPC 接口还是传统的 ISA Bus，而从中来选择不同的 Super I/O 以及 FlashBIOS 芯片来搭配；至于对 BIOS 设计者而言，则需要在 Super I/O 的 Initializing 子程序，以及 FlashPart 烧写子程序做些必要的修改。

但对于大多数软件设计者而言，Super I/O，IR，软盘接口等依旧存在，控制的 I/O 方式也完全相同，根本查觉不出什么差异性。

SM Bus (System Management Bus)

SM Bus (系统组件管理总线) 是一种遵循着 I2C 协议所设计的有两组信号线的总线联系接口，属于低速率接口 (80KHz～400KHz)，供符合 SMBus 的外围零组件检测、定位、读写参数等设置之用。

近代主板由南桥芯片组内建 SM Bus 的控制电路 (SM Bus Controller)，主板能够由 SM Bus，检测 DRAM 插了几排，并自动抓取该排 DRAM 的时序参数 (SPD)，以及读取硬件监控芯片的参数值，来控制 CPU 、主板的工作温度、工作电压 (Voltage)，以及散热风扇的马达转速 (RPM) 等等。

若是笔记本计算机的主板，可以由 SM Bus 检测电池的电力指数，以及外围设备、零组件可独立关闭 / 再激活的可行性，然后通过 ACPI 协议，在整个系统需要进入节约电力的状态下，由 SM Bus 来把目前用不着的外围设备、零组件的电源暂时关闭，以达到电源管理的最佳化。

USB (Universal Serial Bus)

USB 是由许多计算机与通讯工业的厂商，如 Compaq，IBM，DEC，NEC，Intel，Microsoft 以及 Northern Telecom 等公司所共同发展新一代的计算机总线标准，中文翻译为“泛用序列总线”。97 年先导入 x86 主板，随后 99 年 Apple iMac 也采用 USB，在全球引爆并加速 USB 外围设备的普及。

USB 总线提供中低速率外围设备的扩充能力，像键盘、鼠标、摇杆、喇叭、麦克风、调制解调器、摄影机等设备，只要是 USB 接口设计，就可以以热插拔 (Hot Plug) 的方式，直接跟计算机连接或拆除 (离线)，计算机与 OS 会自动检测并启用 / 禁用该设备，达到真正的即插即用 (Plug and Play) 的理想效果。

图2-11 USB连接端口与连接装置

USB 的扩充能力最多可连接 127 组外围装置 (含 Hub)，每个 Hub 与装置的连接距离可达五公尺之远。并且 USB 连接器经过特别设计，不论是安装或拔除，都十分地容易就可以上手。而 USB v1.0/1.1 提供最大 12Mbps (=1.5MB/s) 传输速率，目前主板都实作到 1.1 版的规格，2000 年 4 月 27 日最新定案的 2.0 则提高到 480Mbps (=60MB/s) 的传输速率，预料要等到今年才会有相关适配卡、外围装置的支持，并且成为芯片组的支持项目。

在 USB 制定初期，硬件实作规格上出现了由 Intel 制定的 Universal HCI (Universal Host Controller Interface) 以及由 COMPAQ 设计并开放的 Open HCI (Open Host Controller Interface) ，一般 USB 外围装置都支持这两种协议。以下是它们的差异比较表：

规格 / 差异项目	UHCI (Universal Host Controller Interface)	OHCI (Open Host Controller Interface)
首次发表时间	1996 年 1 月 15 日	1995 年 11 月 22 日
厂商 (IP 来源)	Intel(英特尔)，为专属智财	Compaq (康柏)，属开放性智财
芯片组厂商	Intel (英特尔)，VIA (威盛)	ALi (扬智)，SiS (硅统)
设计特色	电路实作较容易，成本低易整合但大量传输时性能较差	电路设计较复杂，但较不会影响 CPU 与总线性能
控制 / 寻址模式	IO base (IO Port)	Memory address
传输模式	The Frame List is an array of up to 1024 entri	32 period entries ControllHeadED BulkHeadED

由于 USB 驱动方式比较复杂，一般 USB 装置都是进入操作系统下，加载适当的驱动程序来驱动、使用，而且以 BIOS 容量寸土寸金的特性，也不适合将 USB 控制程序也烧写进去；但是随着 USB 键盘、USB 软盘、硬盘与光驱等装置越来越流行，如果有人不接 AT/PS2 规格的鼠标、键盘，硬要直接使用 USB 的鼠标、键盘等装置，BIOS 又怎能不支持呢?

所以新近的 BIOS，直接提供了最基本的 USB 装置驱动与读取功能，比如可以使用 USB 键盘、鼠标，以及用 USB 软盘、硬盘甚至 USB CD-ROM 来开机。微软提倡的 Legacy Free PC ，其 BIOS 就指明了要具备这种 USB Device Boot 的能力。

当然要发挥完整的读写效果，即使只是开机加载 DOS，BIOS 所提供的 USB 功能还是不足，此时还是要靠安装厂商提供的 USB 驱动程序，才能完整发挥各种 USB 装置的读写功能。

GP I/O (General Purpose I/O)

GP I/O (General Purpose I/O) 是从 Intel 430TX Pentium 芯片组之后，近代芯片组提供的通用 I/O 控制信号线，一般都提供十几组输入 (GPI，General Purpose Input) / 输出信号线 (GPO，General Purpose Out)，作为主板上零组件的特殊控制，让系统 / 主板厂商能多发挥创意，为计算机 / 主板创造出许多附加功能。

比如键盘开机 (Keyboard Power On)、网卡摇控开机 (Wake on LAN)、调制解调器 / 传真机来电开机 (Modem Ring On)，以至于近来最流行的 CPU 过热防护功能 (CPU Overheat Protection)、线上调整电压 / 外频 (超频功能，Overclocking)，以及 AC97 主板上的 Codec 芯片如何被检测、启用或禁用，都是运用 GP I/O 来设计相关硬件线路，配合 BIOS 程序代码软硬兼施的搭配，才能有这样多特异功能的出现。

大多数南桥芯片为了节省脚位，会把有些 GPI 与 GPO 信号线跟其他讯号线重叠共享，这就是通过 General Configuration register 与 X-Bus Chip Select register 来设定 / 判别的。

AC'97 (Audio Codec 97)

AC97 是 Analog Component 97 (简称为 AC97)，最早概念由 Intel 96 年开发 NSP 多媒体主板时提出。目前最新版本为 V2.1 (1998 年 5 月 22 日发布)，随后加以延伸扩充成后来的 CNR 1.0 规格。AC97 其规格主要精神是在计算机平台 (主板) 将声卡芯片、调制解调器传输芯片等涉及模拟 / 数字转换的电路功能集成时，将声卡 / 调制解调器中的电路功能分割成两组：模拟译码芯片组件 (Analog Codec) 与数字组件 (Digital Codec)，纯数字译码组件做在主板上，而模拟信号转换部分另外拉成以插在扩充槽上的直立子卡 (Riser Card)，由主板 (的芯片组) 以所谓的 AC-Link 信号线做控制、传送，如此可以降低主板高频信号的干扰，提升内建音效芯片的主板音质。而这个特别的扩充槽，就称为 AMR 槽。

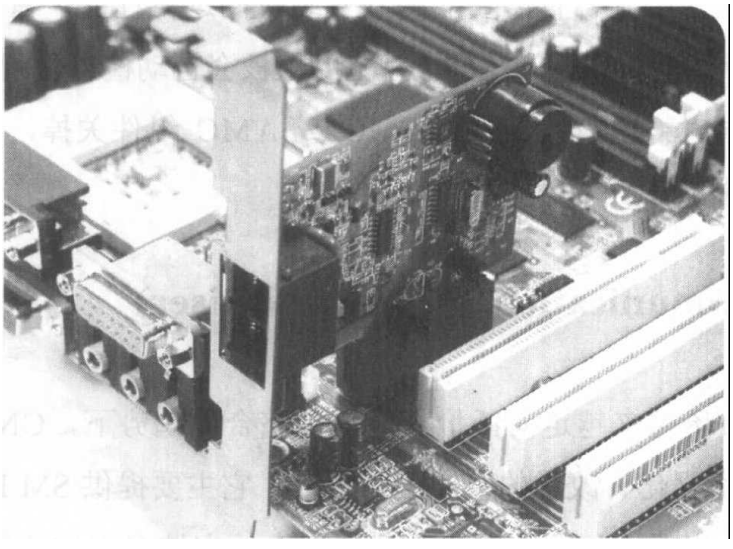


图 2-12　AMR 扩接卡安装的情形

AC97 的规格还有一种特性，可以接符合 AC97 规格设计的低脚位 / 低成本 Audio Codec (AC) 译码芯片，借助 CPU 的运算能力，仿真大约 16 位声霸卡等级的录放音以及 MIDI 音效播放功能 ；符合 AC97 的 MC (Modem Codec) 则可以软件方式仿真 V.90 规格的 56Kbps 调制解调器功能，当然也有可做音效 / 数据传输的 AMC (Audio/Modem Codec) 。Intel 还定义出混音器，直接从传统喇叭 Line-out 输出到喇叭上，所以主板连传统的蜂鸣器都可以省略。

现在最新的主板，多半都符合 AC97 规范，内建 AC 或 AMC 组件，如果通过 BIOS 开启，就等于内建了一个廉价的声卡 / 56Kbps 调制解调器功能，对于一般只作文书打印、只是上网，或者玩非着重声光效果的游戏用户还算足够应付。

不过激活 AC97 的 Audio 或 Modem 功能，都会吃掉 CPU 大半的执行性能与资源，因此使用者若有需要，另外安装较高运算性能 / 音质的声卡，或是外接的调制解调器时，符合 AC97 规范的主板会自动检测使用者外接声卡 / 调制解调器，同时将主板上的 AC，MC 或 AMC 组件关掉，改由硬件声卡 / 调制解调器来接手。

CNR (Communication & Network Riser)

在主板市场越来越走向低价 / 多功能整合的趋势下，CNR 是 Intel 针对 AC97/AMR 所强化、改进的直立子卡规格。它主要提供 SM Bus、两组 AC97 信号线控制，以及一组网卡联机装置的能力，支持多声道环场软件音效仿真，V.90 (56Kbps) 调制解调器、HOMEPNA (家用电话线局域网) 、10/100Mbps Ethernet 网络适配卡之外，还可另外做 USB 接口的扩接槽，以增加计算机系统可使用的 USB 端口数目。

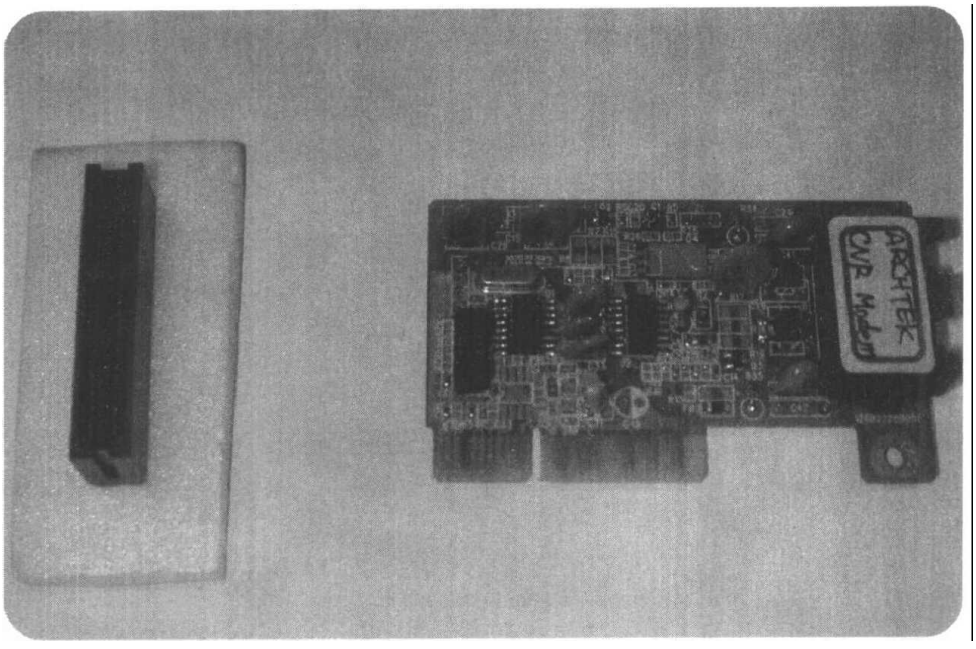


图 2-13　CNR 扩充槽与扩充卡

其扩充槽仍属于咖啡色的两段式，下段比较长，大小也比原先的 AMR 槽还长一点，但设计上连旧的 AMR 子卡也可以插在 CNR 扩充槽上，跟过去的 AC97 保持一定程度的兼容性。

ACR (Advanced Communication Riser)

ACR 是新一代网络 / 数据 / 音效直立子卡的规格，由 AMD 主导，联合当今知名的一级网络、数据通讯、主板与音效芯片组大厂如 3Com，ALi (扬智)，Conexant，Lucent (朗讯)，Motorola，NVIDIA，PCTEL 与德仪 (TI) 等，合组 ACR (Advanced Communications Riser)“先进通讯直立卡联盟”。

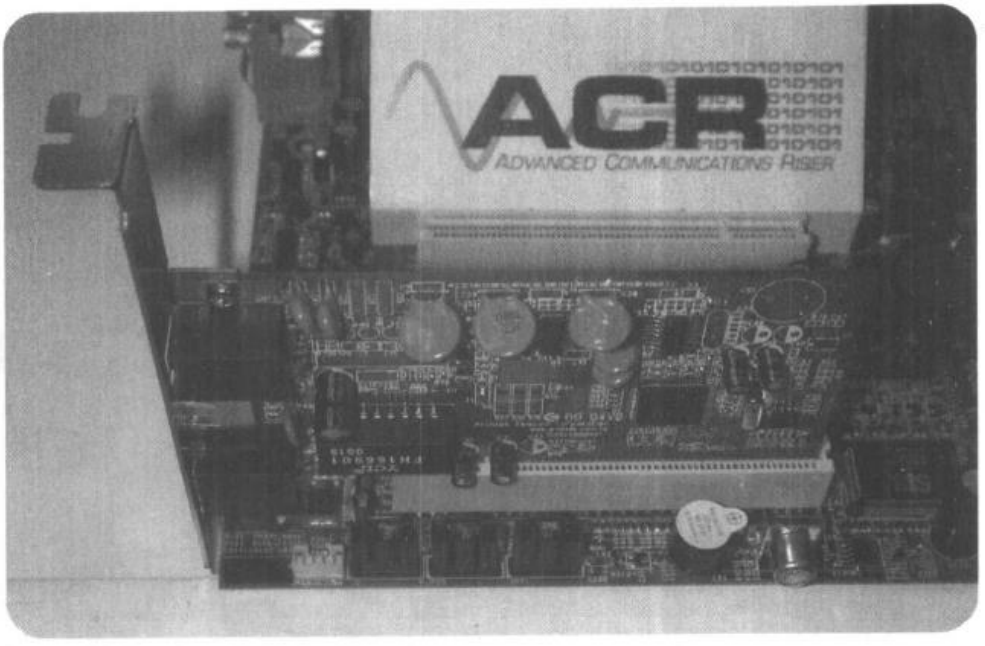


图 2-14　ACR 扩充槽

ACR 子卡的扩充槽摆设，是以 PCI 扩充槽倒转 180 度设计(当然内部信号定义跟 PCI 不同)。它强调能比 Intel CNR 规格还多样化的功能 (多增加 ADSL 软件调制解调器的选项)，以及采用倒转 PCI 扩充槽的成本优势。但是目前支持并实作出来的厂商比较少。

Legacy Free PC

Legacy Free PC 字义是颠覆、跳脱传统结构，以全新的 PC 规格设计的新一代计算机，也是 Intel，Microsoft 等国外大厂积极呼吁的目标。它以千变万化的造型，活泼而赋予创意的空间设计，搭配种种全新定义的软 / 硬件接口规格，将个人计算机的设计带入时尚装饰品、甚至是艺术品的境界。

从主板平台规划来看，它完全不考虑过去传统结构的束缚，如废除目前主板仍然保留的 ISA Bus，ISA 扩充槽，只支持最新、自动检测 / 设定的总线与接口规格。如 PCI，AGP，USB，SM Bus 等，声卡、调制解调器适配卡也逐渐改以 AMR 或新一代 CNR 接口去做。

由于自 Intel 提倡 USB 以来，至今每块主板都已经内建 USB 接口，USB 装置也逐渐普及，1.1 版接口已经普遍流行，每块主板都已经内建。因此传统的 2S1P1G (串行接口、并列端口、摇杆接口) 以及 PS/2 接口等传统中低速率接口完全都废除掉，Legacy 不再有任何传统的扩充 I/O 接口，有的只剩 USB 之外还是 USB，顶多再扩充 PCI，AGP 显示卡，其余就不需要了，这一切都是为了让计算机更容易使用 (Ease of Use，EoU) 而所做的努力。

	Legacy PC (传统PC)	Legacy Free PC
造型设计	较为固定格式、呆板	较为活泼、多样化
主板尺寸	ATX，MicroATX	MicroATX，FlexATX
ISA Bus、ISA Slot	有 (凭借 LPC)	无 (完全废除)
PCI Bus	有	有
AGP Bus	有 (非必备)	有 (非必备)
USB Bus	有 (必备)	有 (必备)
Serial Port COM1&COM2	有	无 (完全废除)
Infrared Port (IR)	有 (非必备)	有 (非必备)
Parallel Port (LPT1)	有	无 (完全废除)
Game I/O	有	无 (完全废除)
USB 装置可直接开机功能	有 (非必备)	有 (必备)

另外一些较为属于技术细节的项目，如不再以传统 RS-232/Printer Port 来作系统除错设计，而是改以新一代的 Debug Port 规格，并由南桥芯片去内建。此外也废除掉像 A20M# 线路的实作，所有主板 / 芯片组默认值就是 A20M# 拉高电位 (Pull High)，也就是默认 A20 的线路开关是打开的。

此时以 DOS 软件的真实模式来寻址，FFFF:0010～FFFF:FFFF 所寻址的内存空间，不会再绕回 000000～00FFEF 的内存地址，而是位于 010000h～010FFEFh，大约是 1024K～1088K (再减掉 16bytes) 的区域。所以 Legacy Free PC 在执行一些 DOS 软件，特别是 I/O 端口直接控制硬件的程序，兼容性因此也就更加受到局限了。

2-3 跟电源管理有关的规格与术语解释

System Management Mode (SMM)

这是 Intel 在开发 386SL，486SL 笔记本计算机特别版本的处理器所增加的操作模式 (一般台式机 486 并没有，随后在 Pentium 处理器以及后期 486 处理器列入而成为标准功能 / 规格之一)，而其他兼容厂商(如 AMD)也在后期 486，K5 处理器时代加入 SMM 模式，因此 Intel，AMD 两家处理器的 SMM 模式，基本上格式都是完全兼容的。

当计算机因一段时间没有操作时，此时通过 BIOS 或操作系统的协助，把外围设备 (硬盘、显示器、键盘、鼠标) 关掉后，CPU 进入 SMM 节电模式，将 CPU 必要的信息存储起来后，CPU 完全停止工作，耗电量也达到最低水平。

SMRAM (System Management RAM)

这是 CPU 进入 SMM 节电模式时，所存储的处理器、芯片组与苏醒 / 回复 (Resume) 子程序码等必要系统信息的工作区域，一般所在的内存地址大约在 030000h，也就是从内存地址最前面起第 192KB 的区域，以真实模式寻址为 3000:0000～3000:0FFFFh，存储空间为 64KB(含前面的保留区、SMM 处理 / 回复程序代码与 512bytes 的系统状态区)，其使用情形如图 2-15 所示，这个地址也可以在 4GB 寻址空间内任意指定，指定的地址称为 SMBase。

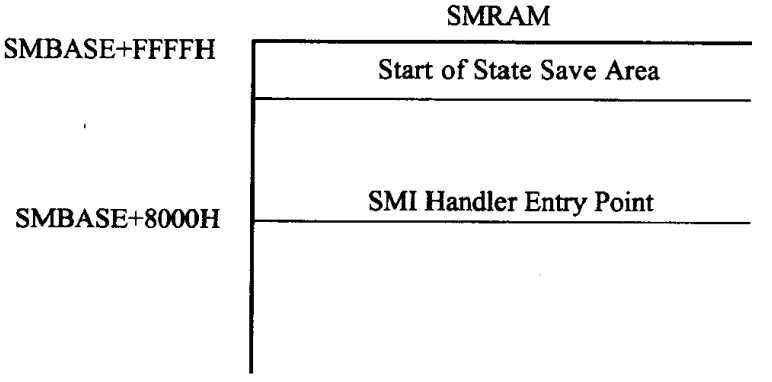


图 2-15　SMRAM 的使用情形

SMI#

当系统进入节电模式，一旦有任何外围装置被触发(如移动到鼠标、按下键盘等)，此时系统会产生一个 SMI# 的硬件中断信号通知并唤醒 CPU，CPU 恢复运转，并且通过 BIOS，OS 取回先前存储的相关信息后，以 RSM 指令恢复到原先暂停的工作状态，如图 2-16 所示。

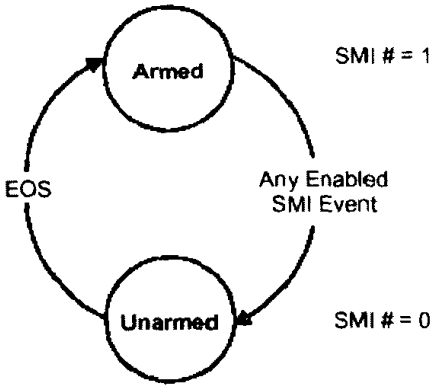


图2-16　SMI工作模式示意图

APM

APM (Advanced Power Management，先进电源管理接口)是Intel与Microsoft在92年所定义的节电管理接口，目前被实作的最新版本为v1.2，BIOS的常驻功能调用为INT 15h，AX=5300h～5313h。APM最早由笔记本计算机所采用，后来推广到台式计算机，现在的BIOS都具备APM 1.2的规格。

简单地说，早期的计算机要做外围硬件的电源控制与管理，必须通过APM 1.2规范，通过BIOS这边来控制；支持APM的BIOS，也负责将自己的各种外围装置，如屏幕、硬盘、软盘、主板系统本身等的子程序，以real mode与16bit/32bit protected Mode提供出来，并且建立适当的调用表格，让其他操作系统(如Windows)也可以直接调用，并控制硬件的电源状态。

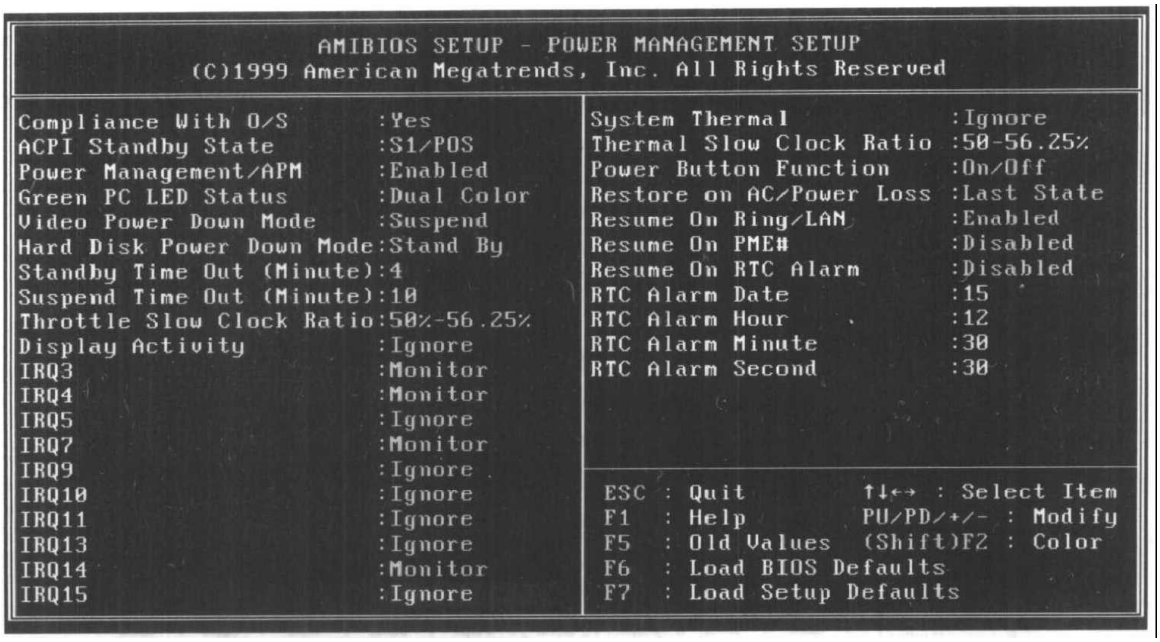


图 2-17　BIOS 的节电管理设置画面

ACPI (Advanced Configuration and Power Interface) 先进电源 / 组态控制接口

由 Intel，Microsoft，Phoenix 以及知名的 Toshiba 笔记本计算机大厂，针对近代 OS 节电需要所重新定义的新一代电源 / 组态控制接口。各家 BIOS 所实作的规格是按照 1999 年 2 月所公布的 1.0b 版本，而最新 2.0 版本于 7 月 27 日公布，比较偏向于 Server 服务器功能的追加，对于一般台式 / 笔记本计算机所增添的功能并不多。

上面介绍过 APM v1.2 电源管理都交给 BIOS，如果你在 Windows 下把电源管理改成 APM 模式，也是把电源管理的大权都交由 BIOS 去管理。由于 APM 定义的功能较局限，能做到的顶多是设置几分钟之后什么装置的关机，并且做简单的监控，电源节省的效率并不是最高的。因为实际开机后，真正主导、控制整台计算机的是操作系统。因此电源管理的作业重心，必须要由 BIOS 转移到 OS，才能真正完全发挥到最高的电源管理效率。

为了更有效率地追踪外围设备的电力使用情况，以及做精确的电源开／关控制，特别是用电分秒必争的笔记本电脑，ACPI 还在操作系统与硬件设备间多加了一个 ACPI 软件／硬件中介层。

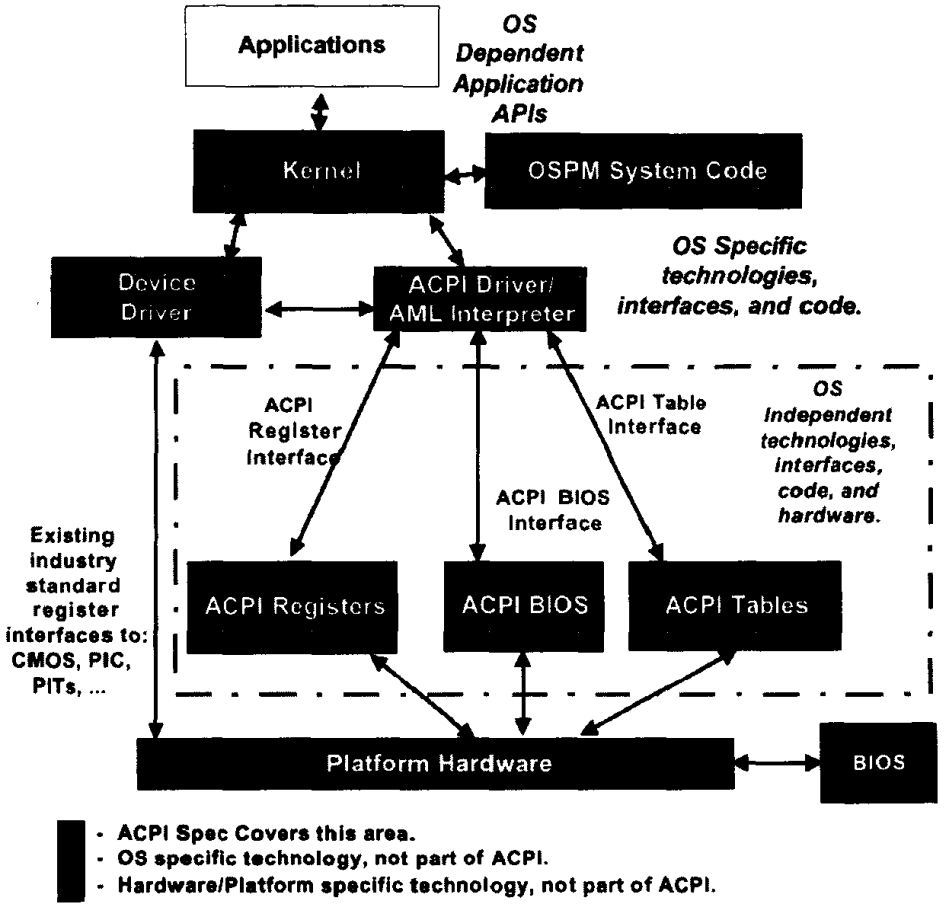


图 2-18　ACPI 运行图

这相当于另外发明一套极为复杂的新控制语言 / 接口协议，一种让外围设备、操作系统都了解的沟通模式。外围设备如软盘、硬盘、光驱、调制解调器、网卡等等，随时将自己的电力使用情形以及开关状况，记录在 ACPI 中介层的寄存器；传感器也随时将最新的电压、温度等参数通过 ACPI，SMbus 传送给操作系统。

而操作系统则是通过 AML 语言解译程序，通过 ACPI 驱动程序对符合 ACPI 规范设计的外围装置，做开启、关闭等动作，或者设定若干时间后自动关闭、开启等等。控制的方式有两种，一种是通过 SCI (System Control Interrupt) 中断信号，另外一种则是通过极为复杂的表格结构，实现外围对外围，外围对 CPU 以及 CPU 对外围的沟通。

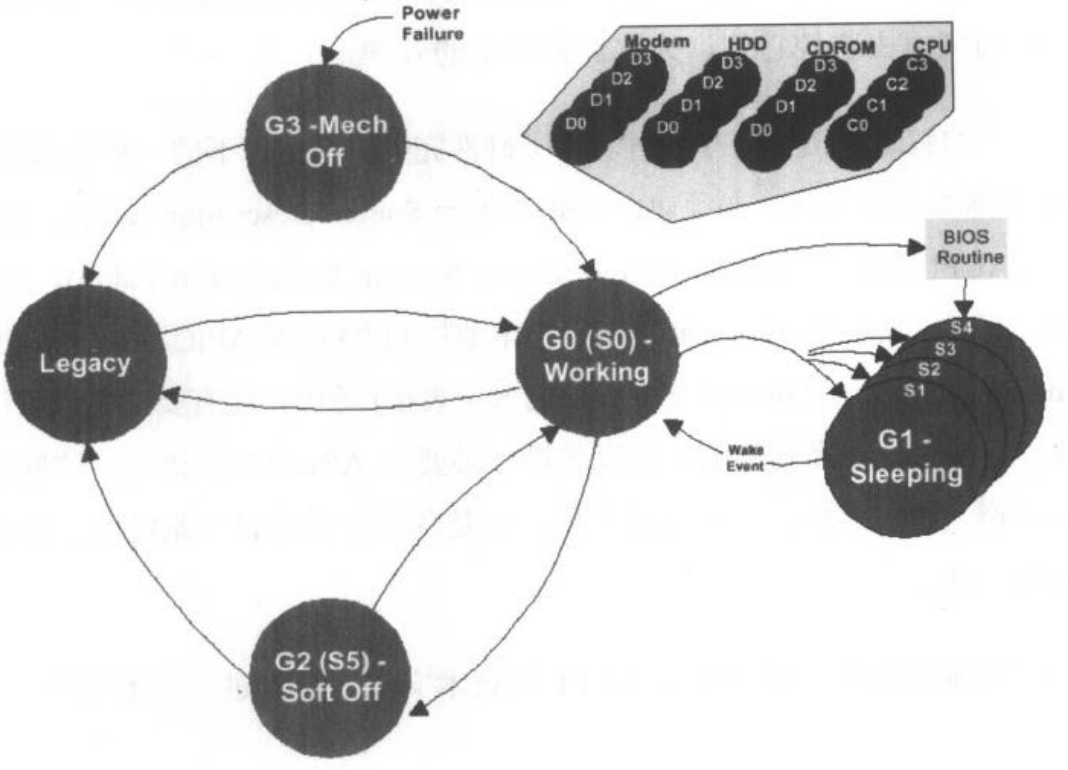


图 2-19　ACPI 所定义的电源控制模式

ACPI 分别针对各种零组件定义了四到五种电源切换模式，如 CPU 的电源使用模式，从轻微省电的 S1，S2，到了 S3 已经是 Suspend to RAM 的模式；而一般笔记本计算机曾见到的 Suspend to DISK (暂停写回硬盘)，已经是 S4 模式，S5 就是真正关掉电源，也不保存原先画面与系统状态的一种模式。

想像一下笔记本型计算机的散热风扇，当检测 CPU 温度上升到一定程度时，会自动打开散热风扇的马达开关，等温度下降再关闭；还有智能型电池的电力指数，也是通过 ACPI 来读取；还有像早期笔记本计算机常用的扩充端口 (Docking Ports)，当插入时唤醒 CPU，PCI 与 PCMCIA Bus，将端口也列入 ACPI 的节电控制。

有些外接的设备 (如打印机、调制解调器) 本身有自己的开关，如果已经关闭了，通过 ACPI，系统可以得知这些设备已经真正被关闭了，也不必再花时间与气力去监控它们，为它们做省电的管理。

以上的叙述,只是 ACPI 种种软件 / 硬件完美协同运作下的一种实际应用。ACPI 系统要正常运作，必须建立 RSDT(Root Source Descriptor Table)，FACP (Fixed ACPI Table)，DSDT(Differentiated System Description Table)，SBST (Smart Battery Specification Table 智能型电池控制表) 以及 APIC (Advanced Programmable Interrupt Controller 中断控制设定表格) 等等，还有要对自己设计的主板、零组件，以及安装的外围设备设计必要的 ASL/AML 语言，这些都是极为琐碎、高技术性、高难度的工程，也是各家开发笔记本系统的工程师最头疼的地方。

有兴趣的读者，可以参阅 ACPI 的技术文献，会有进一步的说明。

ASL (ACPI Source Control Language)

ASL(ACPISourceControlLanguage)源代码控制语言，由一连串可被阅读的指令文字所组成，很像是我们一般写 BASIC 或 PASCAL 的高级语言源代码一样。在各种节电模式(S0～S3) 下，对外围设备的控制程度，更精准到一步一脚印、一条指令一个动作的地步。以下是 ASL 语言的范例：

```
DefinitionBlock (
  "forbook.aml", // Output Filename
  "DSDT", // Signature
  0x10, // DSDT Revision
  "OEM", // OEMID
  "forbook", // TABLE ID
  0x1000 // OEM Revision
   )
{  // start of definition block
   OperationRegion (\GIO, SystemIO, 0x125, 0x1)
   Field (\GIO, ByteAcc, NoLock, Preserve) {
    CT01, 1,
}
   Scope (\_SB) { // start of scope
     Device (PCI0)  { // start of device
       PowerResource (FET0, 0, 0)  { // start of pwr
          Method (_ON) {
           Store  (Ones, CT01)  // assert power   (打开Power)
          Sleep  (30)  // wait 30ms          (等待30ms)
          }
           Method (_OFF)  {
           Store  (Zero, CT01)  // assert reset#  (发出reset 信号)
          }
           Method (_STA)  {
           Return  (CT01)
          }
        } // end of pwr
     } // end of device
   } // end of scope
} // end of definition block
```

AML (ACPI Machine Control Language)

ASL这种纯文本的高级控制语言，其实支持ACPI的操作系统并不识别，也无法直接执行，而且要是直接将ASL源代码文本文件数据都烧进ROM，那ROM的空间可能会不够用，管理零组件电源的独门秘方也容易外泄出去。

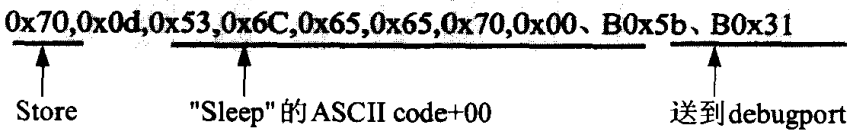
因此，又有一种较为紧密、但仍旧是伪码（无法执行的机器码，或者说它是代表动作的数据代码），称之为AML (ACPI Machine Control Language，ACPI机器控制语言）。一般写出ASL语言之后，还必须借助微软所提供的ASL.EXE (ASL语言的编译程序），把ASL翻译成比较节省空间的AML语言，然后经过压缩后，存储在BIOS的ACPI Table里面。

然后当支持ACPI的操作系统进驻后，它会从ACPI表格中读出AML控制码，然后由OS内部的AML解释程序，将种种AML的控制语句一步一步地解释，然后调用对应的子程序去执行外围硬件的电源（或其他）控制动作。

AML对应于ASL的关系，很类似于.obj (AML) 对应于.asm (ASL) 一样，以下是同样一个动作，分别以ASL与AML语言所表示的结果：

Store ("Sleep",debug)　　ASL与范例，将"Sleep"字符串送到Debug port

经过编译成 AML 之后，是如下这样表示的：



由于 AML 语言比较难阅读、难开发与难维护，对于系统开发者来说，开发 ASL 来作硬件电源 / 系统监控的流程设计，比起用生硬的 AML 语言来编写，要容易许多。

CHAPTER

3

BIOS 开发与硬件之间的互动关系

我们都知道BIOS存储一些开机时的硬件启动/检测码(Initial Code)，也负责一些最基本的硬件外围I/O处理(如屏幕显示、磁盘驱动器驱动、摇杆控制等等)，最后负责从软盘、硬盘或其他装置加载操作系统以完成开机程序。

这样的一组短小精干、程序代码紧凑的BIOS程序代码，有什么样的执行顺序，需要依照哪些硬件特性而有什么特殊的考虑呢？进一步的执行细节又是如何？这就是本章所要讨论的重点。

由于x86 BIOS也是经由十几年的编写与演变，或是为了兼容性的考虑，不可避免地会带有过去硬件结构的种种包袱，本章节在作介绍时，将不可避免地提到一些x86CPU的历史，了解这些因处理器与硬件的时代、规格演变，所赋予BIOS的新规格、新数据结构越多，读者们对BIOS内部的精髓就知道得越详细，故请读者能耐心看完这段BIOS演变的沧桑史！

3-1 CPU历代演变与BIOS的关系

由于CPU世代的演变，除了带来新的高执行速率、新的总线设计以及对外接脚的设计，通常内部的译码结构、执行单元也会有所改变，虽然CPU都有所谓的向前兼容性的考虑，但是新增的功能/规格，往往会设计新的寄存器，有特殊的规划方法，因此CPU进行世代演变时，绝对需要BIOS研发商亦步亦趋的做紧密地支持与配合。

因此我们可以这么说，近代主板(计算机平台)的结构、功能的主要差异，绝大部分取于CPU本身，其次是由刻划主板(计算机平台)的系统芯片组(System Chipsets)所决定。

举个简单的例子，386BIOS 能不能挪到 486 上来使用呢？以新一代 CPU 通常设计成与上一代兼容的特性，只要能够接收以前处理器的指令并顺利执行，照道理并不是不可以；不过 386BIOS 当初在开发阶段，并不知道或不能预料 486 内部结构上的改变 (像是内建 8KB L1 Cache)，每个指令的执行时间可能也会改变 (缩短了)，因此要加入判别新型 CPU 的程序代码，让新版本 BIOS 可以检测到 486 CPU 并开启这个 8KB Cache，同时修改一些 I/O 延时的子程序，才不会因为 CPU 频率、结构的改变，I/O 延时的计时动作也变动了，造成外围装置驱动异常的情形。

因此对于 BIOS 厂商而言，跨代的 BIOS 移植技术难度比较高，同一代处理器平台的 BIOS，比较容易被移植 (Porting)。例如针对 Pentium 主板开发的 BIOS，如果要移植到 Pentium II，可能要花数个月全面改写；但同样是从 Pentium II 主板系统，将其 BIOS 源代码转移到另一套 Pentium II 系统，也许只要花几星期的时间就编写完成并成功地移植过来了。

而同一代又同一家(甚至同一种)芯片组设计的主板平台，其 BIOS 移植的技术难度又更低了，因为同一家的芯片组结构又更接近。例如 BIOS 厂商跟某 A 大厂合作，开发好了针对 Pentium II /III处理器，而且主板是采用 Intel 440BX 芯片组设计的，那么另一家 M 厂商也有类似平台的 BIOS 开发需要时，通常只需要一星期不到就可以开发 / 移植过来。

Intel 从 Pentium 处理器时代 (以及 486DX2 的后期)，加入一个称为 CPUID 的指令，可以读出目前正在执行的 CPU 各种详细的信息，包含 Family(家族代号)，Model(型号)，Stepping (批号)以及其他关于 L1，L2 高速寄存的信息，供 BIOS 程序代码判断并做对映的初始设置，另外也可通过 Pentium 处理器时代追加的 Time Stamp Counter(TSC)时序记数器，来判断目前处理器的运作频率是多少。

像 96 年推出的 Pentium Pro 的处理器，它具备一项称“微码更新”(Microcode Update) 的功能，万一某一批集成电路在设计当初有没被检测到的瑕疵 (Bug) 时，只要这个错误还可以由微码的方式加以修正，BIOS 就可以通过 Microcode Update 的功能，将 Intel 所送来的某批号的修正微码，在开机 POST 阶段就写入 Pentium Pro 内部的 MicroROM，这样就可以“修下 CPU 的 Bug ”，而 Intel 也就不需要大费周折地回收有瑕疵的 CPU 了。

所以老实讲，现在的新一代 CPU 的 BIOS 研发，已经不像以往那么困难，真正 BIOS 移植技术上的困难度，反而都落在厂商所采用的芯片组、Super I/O 芯片的差异，厂商附加功能上的技术支持；甚至连芯片组、主板在某个时空、某个批号的品质稳不稳定时，也需要 BIOS 这边以软件的方式加以解决。

在此我们提到一个题外话，Transmeta (全美达科技) 所推出的 Crusoe 处理器，其核心结构跟你我熟知的 x86(从 8086/88、286、386、486 到 Pentium、Pentium II/III...)结构有很明显的不同，它是 VLIW(Very Large Instruction Words 超大指令字结构)，但是它又可以拿来设计 x86 的笔记本电脑的处理器，或者强调低耗电的 IA 装置的运算核心。

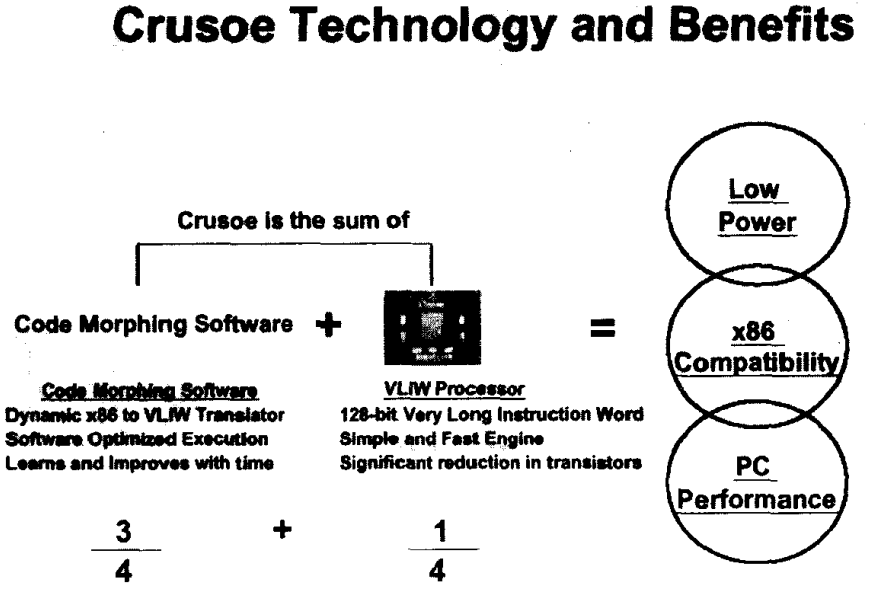
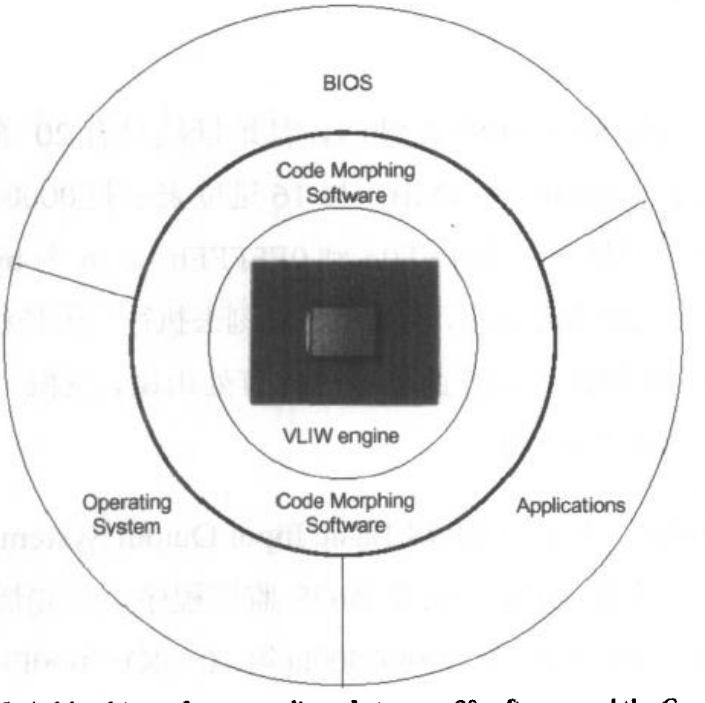


图 3-1 Transmeta Crusoe 的运作结构 (取自 Transmeta 网站技术文献)

Transmeta Crusoe通过一种称为 Code Morphing 的技术，以极有效率的 x86 转 VLIW 指令集转译 / 编译技术，直接编译传统 x86 指令码并且翻译出对等的 VLIW 机器码执行，虽然执行效率会略微偏慢，但可以由提高频率、减少晶体管数量 / 设计面积来弥补，这也就是完稿之前，Transmeta 开始强调低耗电的笔记本电脑、便携式微型 IA 装置的理由。

有趣的是Code Morphing技术是在操作系统层次运行呢？还是在BIOS层次运行，据一些负责研发 Crusoe 处理器 BIOS 的工程师表示，Transmeta 的 BIOS，除了极少数的修改 (像 CPUID、初步开机的 initializing) 外，其他几乎都直接沿用原先的 x86 结构的 BIOS 程序代码，或许我们可以这么说，Code Morphing 技术是一套已经写在 MicroROM 内部的 MicroBIOS，它负责做 x86 机器码转 VLIW 机器码的动作，对于最外层的 BIOS 程序、OS 以及应用程序，Transmeta Crusoe 处理器还是一个地地道道的 x86 兼容型处理器。



The Code Morphing software mediates between x86 software and the Crusoe processor.

图 3-2　Code Morphing 示意图 (取自 Transmeta 网站技术文献)

3-2 x86CPU 内存结构与 ROM 大小的演进

Intel 设计的 8086/8088 处理器，CPU 开机后的第一个动作，就是 CS=DS=ES=SS=0FFFFh IP=0000h，将指令定位器指向 FFFF:0000h，换算成物理内存地址(Physical Address)，就是位于 000FFFF0h 的区域，然后开始获取指令码执行。随后 80286，80386，486，Pentium 等 x86 处理器，以及 AMD，VIA/Cyrix，Rise 等兼容型 x86CPU，也都保留了这样的设计特色，这是基于硬件向前兼容性(Backward Compatibility)的考虑。

这种 CPU一开机，就从内存最高地址获取第一道指令的硬件设计，在许多处理器设计上都可以发现。而 x86CPU 的这种特性，决定了它 BIOS 所存储的 ROM 芯片，必须被限定为能译码这个最高内存地址；BIOS 程序代码在编写上，也必须依照这个特定的内存位置，甚至在某个阶段还有不同限制的法则，才能一步一步地唤醒整个系统零组件，顺利地完成运行，并且将控制权引导到操作系统。

显而易见，最早 8086/8088 处理器，其地址线只有 20 条，最大存储器的寻址空间只到 2^{20}=1024KB (=1MB)，以 16 进位表示是 00000h～FFFFFh 这个区域，从 CPU 一开机时的 0FFFF0h 到 0FFFFFh 仅 16 个 bytes 的空间，根本不可能放下任何足够的起始码，因此除了立刻去执行一道长程跳跃指令跳走之外，别无他法。就像机场跑道前方不远处有处山崖，飞机一起飞除了掉头转向之外，别无其他的办法。

大家都知道，第一个创造 BIOS(Basic Input/Output System)这个名词的是当时被称为蓝色巨人的 IBM 。它将 BIOS 监控程序以预先烧写的方式写入 PROM 芯片，然后译码到位于 000FE000h(第 1016K)～000FFFFFh(01024K) 共 8K 的内存区域，这个地带也是存放 XT BIOS 最精华的区域，在最初里面所定义的许多子程序调用点或跳跃点，由于 IBM 将 BIOS 的数据结构完全

公开，使得后来 BIOS 为了兼容起见，也都刻意保留了这样的区域，以及几乎相同的数据结构与子程序调用点、跳跃点地址。例如前述的 0FFFF0h 地址，一开始存放的机器码依次是 0EAh，05Bh，0E0h，00h，0F0h，也就是 JMP FAR PTR 0F000:E05Bh，跳跃到 F000:E05Bh(或者说 000FE05Bh) 处，您可以发现几乎每一家 BIOS 在这个部分设计都是一样的，如图3-3所示。

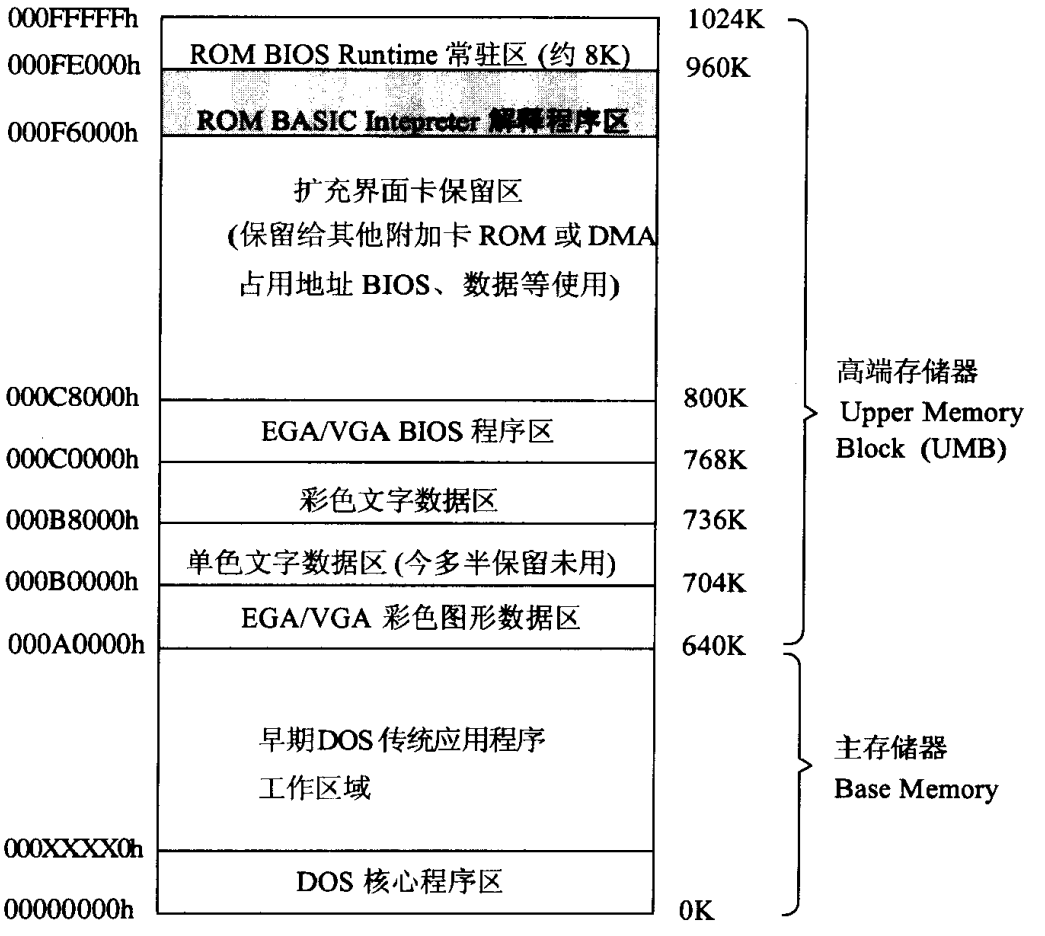


图 3-3　最早期 IBM PC、PC XT 时代的 BIOS 与内存结构

网点区域仅限于 IBM 原厂 PC 系列，非 IBM 原厂的兼容型 PC，XT 并没有 F600:0~FE00:0这一段内存区域

而 0F6000h～0FDFFFh(第 984K～1016K, 共 32K)内存区域，原先是保留给 ROM BASIC 语言解释器程序的，但其他开发兼容 BIOS 的厂商，由于版权或其他因素不能使用这个区域，事实上它们并没有实作这一段，而是以空白或填充其他程序代码方式交代过去，再加上当时微软也推出不需依赖 ROM BASIC 的 GWBASIC 解释器程序，所以 0F6000h～0FDFFFh 这一段欠缺，也不会造成什么兼容性的问题，除非是厂商写了只能在 IBM PC 执行的 BASIC 程序，需要检查这个内存区域，一般这种“自我设限”的程序写法，除了 IBM 本身以外，别人是不可能这样去开发的。

286AT 时代，由于 286 CPU 功能的增加，以及加入 CMOS/RTC 的 CMOS 计时芯片的机制，ROM BIOS 的长度又飞快的增加，从 0F0000h～0FFFFFh (第 960K～1024K)全都使用上，但是此时 BIOS 扣掉 0F6000h～0FDFFFh 这块 32K 给 ROM BASIC 解释器程序，实际使用仅 32KB 不到，而且被分成两大块，位于 0FE000h～0FFFFFh 顶端 8KB 的传统 XT BIOS 区段，提供文字显示、软盘读写的基本功能，以及 F0000h～F5FFFh 这段给 AT 使用的扩充段 BIOS，则补足并提供较为齐全的 BIOS 全方位功能。

另一个影响至今的主 BIOS 区段结构，就是 Phoenix 公司以 286AT 为平台所开发的 286 兼容 BIOS，它所使用的内存区域，除去有版权争议的 ROM BASIC，改寻址于 0F8000h～0FFFFFh，0FE000h～0FFFFFh 这顶端 8K 仍旧是传统 XT BIOS 区段，这一段摆放基本磁盘驱动器的读写功能、必要的 POST 程序代码与 Initial 码；至于 0F8000h～0FDFFFh 一共 24K 的区域，被定位成 AT 扩展的BIOS 区段，提供完整的 BIOS 功能与函数调用。

位于 0FE000h～0FFFFFh 这部分程序区段，为了跟以往的 IBM PC XT/AT 达成撤消性兼容，程序结构、流程、跳跃点与若干变量参考几乎已经固定了，一般 BIOS 版本做日后功能上的追加或修改时，几乎不会动这个部分，而且一般 BIOS 芯片在做 Reflash 时，也都针对这个区域有特别处理。除了特别指定，是不会随意变动这个区域的程序代码。这一段程序代码，业界一般也俗称为“激活区段”(Boot Block)，因为它是 CPU Reset 后的“必经之地”。

在 286AT 之前的 BIOS，CMOS 设置程序是分离在 BIOS 之外，设计成放在 DOS 磁盘上的工具程序，不过如 Phoenix BIOS 或后来出现的 AMI，AWARD 等，并没有把 0F6000h～0FDFFFFh 放入 ROM BASIC，而是跟微软合作开发 GWBASIC。所以当时兼容 PC 的 BIOS，可能只有 0FC000h～0FFFFFh 约 16K 的空间，最多 32KB。

进入 386 平台之后，由于产业规范增加，PROM 内存芯片容量增加、价位下跌，因此 386 时代的 BIOS，已经把 CMOS 设置程序整合进来，再加上支持各种外围装置，使得 BIOS 的程序代码迅速膨胀到 32KB，有些版本已经要占用 0F0000～0FFFFFh 一共 64KB 的区域。

接下来 486 时代 CPU 百家争鸣，加上新形态的 IDE 硬盘容量限制，LBA，CHS 的转换，新型芯片组的出现，以及 PCI 总线的流行，BIOS 程序代码此时已经从 32KB 突破到 64KB 大关了！随后美国环保署定义出环保计算机的规范，因而有所谓的 GPC(Green PC)以及往后 APM(Advanced Power Management)规格的推出，也在 BIOS 内实作出来。

大约是486后期甚至Pentium时代，BIOS开始导入模块化(module)的概念，这样在针对新功能、新规格时，只要开发新的模块后加入即可，原先的模块为了兼容性起见能不动就不动。但是ROM BIOS的程序代码长度越来越大，64KB实在是不够使用，因此在Pentium中期导入了“模块压缩/解压”的概念，BIOS只要有几个不压缩的程序代码模块(包含解压缩模块)，在烧写时先将不是马上要运行到的模块压缩起来，要调用时解压到Shadow RAM或PCI Memory，而这种解压缩的机制，每一家BIOS厂商都实作出来，以求在寸土寸金的EPROM芯片，塞入功能最多、支持最齐全的BIOS，以争取客户的信赖与认同。

至此ROM BIOS长度停留到64KB～128KB有一段很久的时间，每一家主板厂商会依自己的主板市场定位、想支持的模块功能，以及当时零组件的成本，来决定要不要增加这些功能。

随后Windows 95时代导入PnP规格，而这部分需要借助能够重复擦除、写入的Flash EPROM，加上ACPI，USB接口的起始码，BIOS即使压缩也很难再维持在128KB以下(除非删除掉部分功能)，就笔者所知大约进入Pentium II，Athlon处理器时代，非整合芯片组主板BIOS已经是256KB，若是像810e/815e内置了显示芯片的主板，其BIOS容量已经需要512KB了。

MemorySizing(内存容量检测)与A20开关与保护模式切换

各位也许不晓得，在IBM PC与XT时代，计算机的日期/时间是开机时要事先输入的，要不然一律从1980年1月01日开始，随后有许多外围接口卡厂商开发了计时芯片，也运用低耗电的CMOS芯片来存储先前记录的系统时间、日期，随后IBM终于在AT时代正式导入了CMOS/RTC芯片。起初CMOS可索引、记忆的地址仅64组，后来增加到128组，一些像年代、日期、时间等CMOS索引地址都固定了，其他计算机厂商为了兼容性考虑，也参照这些公开的索引地置来设计BIOS，至于其他未使用的部分，就给各BIOS厂商自己分配使用啦。

我们知道286的内存地址线高达24条，它的最大内存寻址为2^{24}=16，384KB=16MB，但是要涉及这个模式，需要将CPU设置成保护模式(Protected Mode)，CPU一开机时以及Reset会进入真实模式，真实模式就是沿自8086/8088的segment:offset的内存操作模式，如果以Segment=FFFFh，offset偏移值从0010h～0FFFFh来寻址，最大能表示的内存区域是FFFF:0010～FFFF:FFFFh，内存寻址空间超过20条地址线的表示，变成从100000h～10FFEFh(约1088K扣除16个bytes的空间)。

而以往8086/8088以Segment=0FFFFh，当内存地址超过0FFFFFh时会绕回000000h，因此FFFF:0010h～FFFF:FFFFh等于又绕回00000h～00FFEFh，因此286以上的CPU，在地址线A20～A23(386以上则为A20～A31)设计了一个开关，这就是所谓的A20开关：可以强制归零或可进位。当强制归零时，就可模仿8086/88的内存地址回绕的特性；若设定可进位，则可以在真实模式下，多出大约64KB的内存空间。记得很久以前DOS 5.0版刚推出时，就是运用了这个内存空间存放部分的DOS核心程序，将传统640KB主存储器空间空出来，以执行较大型的DOS软件包。

如果要搜寻 1MB(A20 off)或 1MB+88K(A20 打开)以外的内存区域，像是检查有没有安装扩展内存(extended memory)，以及确定安装了多少容量的 DRAM 内存(这个动作称为 Memory Sizing)。此处就发生一个问题：BIOS 正处于 POST 的执行状态，要检查 DRAM 的数量得进入保护模式，但由于 286 的硬件特性的缺憾，从真实模式切换到保护模式后，除非重新 reset，要不然无法切回真实模式；但 Reset 后不就又跳回 0FFFF:0，BIOS 重头再执行一次，岂不陷入重复循环了吗?

因此，在当时IBM借助一个强化过的嵌入式控制器8042,它等于是一块简易、小型的 CPU，也有自己的专属监控程序，来负责键盘、A20 开关与 Reset 重置硬件的任务，一般我们又把这块 8042 称为 Keyboard Controller (KBC)。286 BIOS 要做 Memory Sizing 时，首先将特定的当机返回指示字节、返回地址写到 CMOS 内存，以及 40:67h (低字节)、40:69h (高字节)，然后下命令 8042 打开 A20 切换到保护模式，再来做 Memory Sizing。

Memory Sizing 做完之后该怎么办呢，接下来 286 BIOS 下命令通知 8042 帮忙 reset 整台计算机，同时关掉 A20；而这段时间内 8042 本身有自己的监控程序，不受 286 系统 Reset 时总线停止工作的状态限制，仍继续工作，接下来 BIOS 一样从 0FFFF:0000h 获得第一道 x86 指令，由于 BIOS 做了简单的 CPU 检测后，就会从 CMOS 读入刚刚存入的 CMOS 当机返回指示字节，来判断这次是真正硬件刚开机、Reset 的状态、还是软件重开机(Soft reset)，若是软件重开机，表示刚刚是切换到保护模式检查完内存后切回真实模式，接下来就读取返回的内存地址并继续执行，这样就可以略过 (bypass) 不需要重复执行的 BIOS 程序片段，快速地返回原先的 POST 程序片段继续进行。

当以后的386CPU出现时，它已经设计了可以从保护模式返回真实模式的机制，386以后的BIOS，不见得得用reset来往返真实→保护模式的方式，但是基于兼容性的因素，还是保留了A20切换、Software reset等机制，而386 BIOS多少还是会做Shutdown Bytes的填值动作，要用老方式还是新方法，那就要看BIOS程序设计者的功力，以及要做到多兼容来决定了。

其实Memory Sizing还有许多要处理的详细程序，如激活Refresh DRAM timer(DRAM刷新时间的间隔／定时器填值设定)、每一组DIMM(Bank)的内存类型与起始／扩展的内存位置，以及决定要送出的电流强度(Drive Current)等等，笔者将在往后的几个章节中陆续介绍。

ROMCS#，Shadow RAM(影射内存)

x86CPU从286时代，地址线一举从原先的20条增加为24条，内存寻址空间也突破1MB(1024KB)，所以ROM芯片的实体译码地址，都是位于最大寻址空间的内存边界(16MB的边界0FFFFFFh往下计算)到64KB或128KB的区域(根据ROM芯片容量而定)。前面也介绍过，x86CPU一开机时是真实模式，没有办法寻址到1MB+64KB以外的内存区域，因此放在ROM的程序代码，如果没有通过一些特殊硬件线路的帮助，无法被读取／执行到。

前面介绍过A20地址线开关的机制，可以将最高的A23～A20地址线基准位降为0，形成“0F0000h～0FFFFFh与0FF0000h～0FFFFFFh内存地址重叠译码”的现象，因此原本位于FFFFFFh处的ROM BIOS程序代码，两者都指向同一组内存区域。

后来由一家 Chips & Technoloqies(后来为 Intel 所并购)开发的 NEAT 芯片组的出现，当主板插上 1MB(1024KB)以上的内存时，它可以将部分内存对应到传统 640KB～1MB 之间的未使用的内存区域 (此时扩展内存数量会减少 384KB)，也就是把 C0000h～FFFFFh，C，D，E，F 四个 segment 根据需要，设置成可只读 / 可擦写的内存区域，每一个对应的内存页大小最小为 16KB (4000h)，这种特性就称为 Shadow RAM (影射内存), 如图3-4所示。

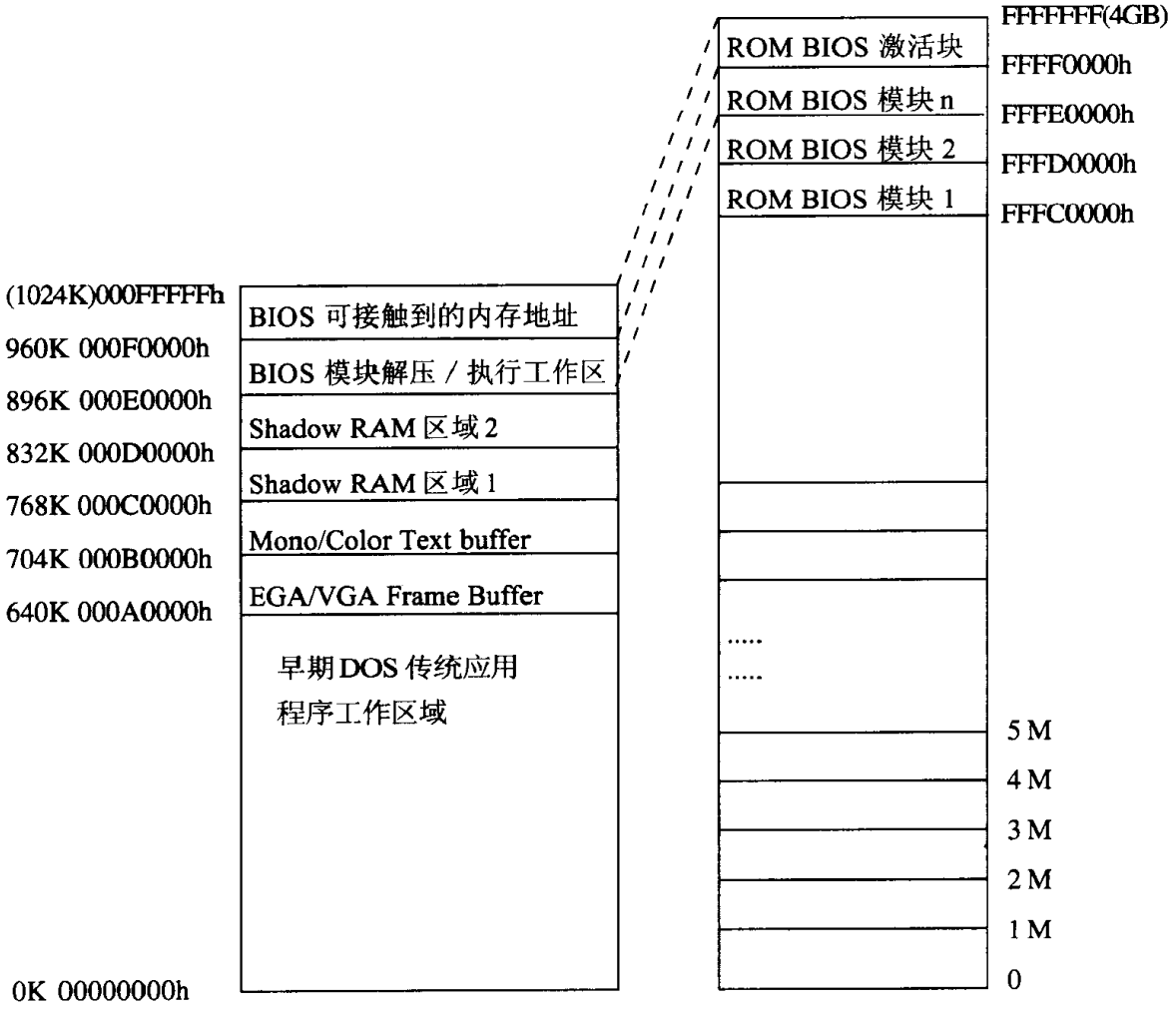


图 3-4　Shadow RAM 示意图

Shadow RAM 的好处有：

1. 将主板 ROM BIOS、显卡与适配卡 BIOS 程序代码，从速度比较慢的 PROM，EPROM 或今日的 FlashEPROM 拷贝到对应的 Shadow RAM，由于RAM的读取／执行速度比ROM还要快，因此可以缩短计算机POST与开机的时间，以及DOS，Windows 或其他 OS 在调用 BIOS 时的执行速度。

2. 配合BIOS模块化／压缩化的观念，存储在实体EPROM/FlashEPROM芯片的 BIOS 程序代码，除了 Boot Block 之外，可以将其他模块先压缩，日后随需要解压到Shadow RAM后再执行。

3. 配合一些内存管理程序，可做类似 UMB(Upper Memory Block)高端存储器、EMS 内存的仿真。

在 Shadow RAM 的机制中，F段(0F0000h～0FFFFFh)的对映方式又比较特别，例如有个控制信号被称为 ROM CS(ROM Chip Select)，意思是指当 CPU 内存地址动用 0F000:0000h～0F000:FFFFh 时，如果有针对该内存区域作写入数据的动作，实际上会真正地写到 4GB 的边界，也就是 FFFF0000h～FFFFFFFFh 的物理内存地址，即原本 FlashEPROM 芯片的实体译码地址。

如果此时 FlashEPROM 的写入电压有提升的话，或者已经被设置成可擦写状态，那么数据就会真正地写进 FlashEPROM 芯片内，并且永远的生效，而这就是目前BIOS烧写工具程序的运行原理。

FlashEPROM 所存在的真正实体地址

以当今寻址能力为 4GB(2^{32})的芯片组而言 (仅服务器等级的芯片组才能寻址超过 4GB)，FlashEPROM 芯片所译码 / 存放的物理内存地址(Physical Address)，均从 4GB 的边界(0FFFFFFFFh)往低地址方向向下扩展，接下来是 APIC(Advanced Programmable Interrupt Controller 可编程控制器) 的内存对应区域，接下来是 PCI 组态地址(PCI Config)、PCI IO 地址、以及 PCI 组态外围组件所占用的内存(当然也包含 PCI 附加适配卡 ROM 所对应的内存地址)，大约在 1.5～2GB 左右(可能更低，视使用者安装到多少容量的 DRAM)，才是真正 DRAM 能够安装到的边界所在。

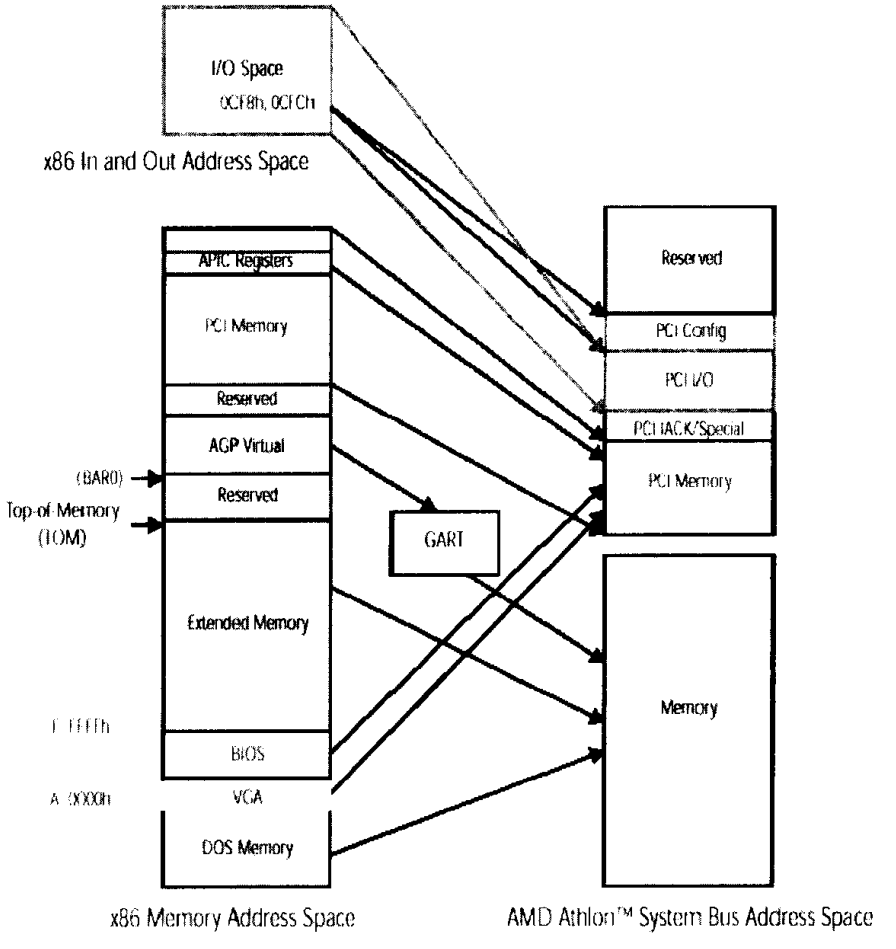


图3-5　最新 x86 计算机平台 (具备 AGP) 内存结构图 (以 Athlon 为例)

而据笔者所知 Windows 系统默认的最大的内存逻辑寻址空间，刚好是00000000h～07FFFFFFFh，2GB 以上(80000000h～FFFFFFFFh)的逻辑内存地址，都保留给系统所使用，也许就是因为系统结构的限制吧！

Flash EPROM 芯片容量与其存放的地址

Flash 芯片容量	物理内存地址	可对映到 1MB 内的内存地址	Boot Block 大小
64KB	FFFF0000～FFFFFFFFh	000F0000～000FFFFFh	8K～16KB
128KB	FFFE0000～FFFFFFFFh	000F0000～000FFFFFh 000E0000～000EFFFFh	8K～16KB
256KB	FFFC0000～FFFFFFFFh	000F0000～000FFFFFh 000E0000～000EFFFFh 000D0000～000DFFFFh 000C0000～000CFFFFh	8K～32KB
512KB	FFF80000～FFFFFFFFh	000F0000～000FFFFFh 000E0000～000EFFFFh 000D0000～000DFFFFh 000C0000～000CFFFFh	64KB

每种 FlashEPROM 芯片依型号不同而有所差异，详情请参阅该 FlashEPROM 的Datasheet

由于寻址空间远超过 1MB (或 1MB+64KB，打开 A20) 之外，CPU Reset 是进入真实模式，根本无法读写该段区域，但是又顾忌跟 DOS 程序的兼容性，所以 AMI BIOS 会运用一种称为 32bit Big Mode 的寻址模式，它的原理是先切换到虚拟保护模式 (Virtual Protected mode)，选择一个 Data Selector (DS，ES，FS，GS)，把它的段极限 (Segment limit) 从原先的 64KB (FFFFh) 加大到 4GB (=FFFFFFFFh)，同时索引地址调成 00000000h，接下来再切回真实模式时，选择这个 Segment 并且把它设为 0000h 的情形下，可以用 32 位的索引寄存器来寻址，而不会造成系统出现错误中断，在整个 4GB 空间内任意漫游。

所以近代 BIOS 在获取 FlashEPROM 的数据时，事实上都是借助这种 32bit Big Mode (或称 Flat Mode) 的寻址模式，这样才能从高高在上的 FlashEPROM 内读取里面被压缩的 BIOS 模块，然后解压到 1MB 寻址空间内的 C0000h～FFFFFh，当然程序代码仍旧是受真实模式 1MB+64KB 的限制，事实上也不需要为了单纯的程序代码拷贝，而大费周折地设定 GDT，LDT 或 IDT，那样的工程过于浩大，快要等于写一套简单的保护模式的小型操作系统 (DOS Extender) 了。

3-3 BIOS 的诊断码 (Post Code) 简介

BIOS程序代码要进行的作业可说是千头万绪，在系统开发的阶段，甚至还没加载操作系统之前，往往主板可能因为某个零件的运作失常、BIOS 程序代码的编写或移植失误等因素，开发过程中常常会卡在某一个关卡无法顺利开机。可是受限于BIOS 程序代码短小，操作系统没加载之前又没有办法执行 debug 之类的软件来除错，只得从茫茫程序海(SourceSea)中搜寻可能出错的程序代码，或者用检测仪器拼命寻找硬件错误的原因。

为了加快系统的开发与除错速度，缩小问题发生的范围，起初IBM 设计 PC 时，就定义出 port 80h 为 Post/debug Port：当BIOS 程序代码要进行某一阶段的作业时，先将该作业所代表的Debug/POST code数值直接送到80h这个I/O端口，系统研发商有专门的除错设备，能够拦截并显示出这个诊断码的数值，在新的诊断码送出之前，这个诊断码的数值会一直保持着。这种简易的 POST/DEBUG 机制，就被以后走PC 兼容的计算机 / 主板厂商所“沿用”了。

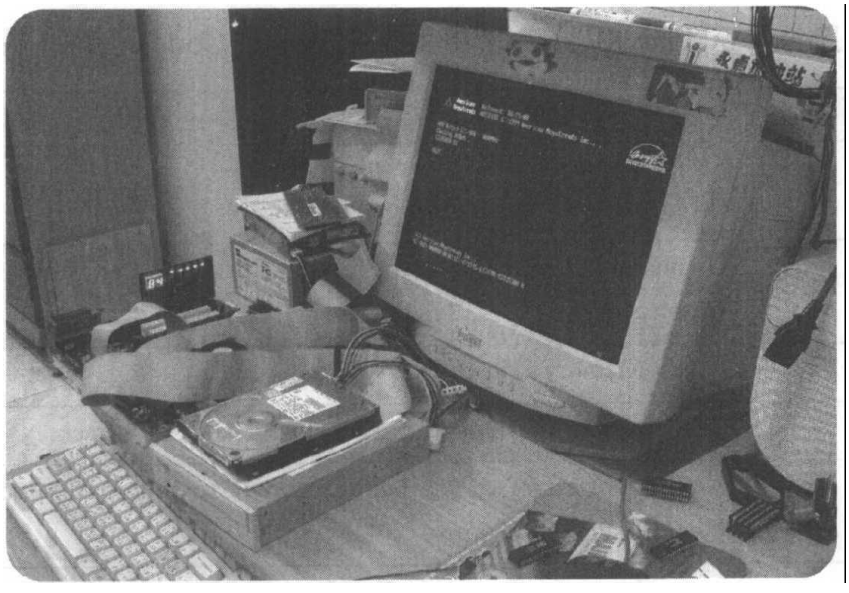


图3-6 笔者的 Porting 主板平台与 Post Card

因此只要我们发现主板在开机加载操作系统之前，诊断码停在哪一个代码，就可以对照源代码或该错误代码的发生原因，直接对症下药找出问题的根源并加以解决。

读者应该看过或听过所谓的 Debug Card (或 POST Card) 除错卡之类的小型适配卡，它的原理非常简单(具备简易电路知识的都会自己制作)，就是针对 Port 80h 译码，并且将送出的数据给 Latch 住，并且通过数字 8 的 LCD 显示屏，显示出目前对 Port 80h 所送出的 Post code 数值资料。

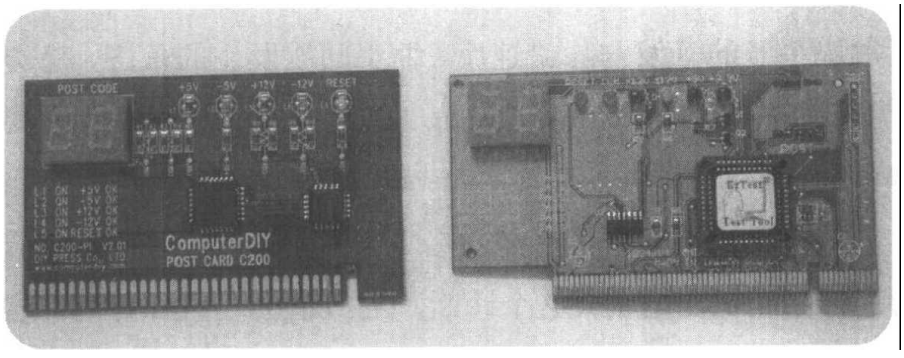


图3-7　ISA 接口 (左) 与 PCI 接口的 POST Card (右)

接下来我们根据 AMI BIOS 所定义 的 POST/Debug Code 列举如下:

诊断码(Hex)	所进行到的程序代码流程、停止或错误发生原因
D0	NMI is Disabled. CPU ID saved. Init code Checksum verification starting.
D1	To do DMA init, Keyboard controller BAT test, start memory refresh and going to 4GB flat mode.
D3	To start Memory sizing.
D4	To comeback to real mode. Execute OEM patch. Set stack.
D5	E000 ROM enabled. Init code is copied to segment 0 and control to be transfered to segment 0.
D6	Control is in segment 0. To check <CTRL><HOME> key and verify main BIOS checksum.

当 BIOS 检测到用户按下 [redacted] + [redacted] 组合键，或者 BIOS 检测到加总检查值 (Checksum) 错误时，会从软盘读取一个名叫 AMIBOOT.ROM 的ROM BIOS 文件，然后进行修复烧写(Flash)的作业，作业过程显示 0E0h～0FFh 的 POST Code；若 POST 过程都没有按下 CTRL+HOME 组合键，BIOS 的 Checksum 也没有错误，则 POST Code 开始跳到 03：

诊断码(Hex)	所进行到的程序代码流程、停止或错误发生原因
D7	To pass control to Interface Module.
D8	Main BIOS runtime code is to be decompressed.
D9	Control to be passed to main BIOS in shadow RAM.
E0	OnBoard Floppy Controller (if any) is initialzed. To start base 512K memory test.
E1	To initialise interrupt vector table.
E2	To initialise DMA and interrupt controllers.
E6	To enable floppy and timer IRQ, enable internal cache.
ED	Initialize floppy drive.
EE	Start looking for a diskette in drive A: and read 1st sector of the diskette.
EF	Floppy read error.
F0	Start searching AMIBOOT.ROM file in root directory.
F1	AMIBOOT.ROM file not present in root directory.
F2	Start reading FAT table and analyze FAT to find the clusters occupied by 'AMIBOOT.ROM' file..
F3	Start reading 'AMIBOOT.ROM' file cluster by cluster.
F4	AMIBOOT.ROM file not of proper size.
F5	Disable internal cache.
FB	Detect Flash type present.
FC	Erase Flash.
FD	Program Flash.
FF	Flash program successful. BIOS is going to restart.

以下是当 POST 过程都没有按下 [redacted] + [redacted] 键时，BIOS 的 Checksum 检查无误时，POST Code 从 0D7h 跳跃过来的 POST Code 显示阶段：

诊断码 (Hex)	所进行到的程序代码流程、停止或错误发生原因
03	NMI is Disabled. To check soft reset/power-on.
05	BIOS stack set. Going to disable Cache if any.
06	POST code to be uncompressed.
07	CPU init and CPU data area init to be done.
08	CMOS checksum calculation to be done next.
0B	Any initialization before keyboard BAT to be done next.
0C	KB controller I/B free. To issue the BAT command to keyboard controller.
0E	Any initialization after KB controller BAT to be done next.
0F	Keyboard command byte to be written.
10	Going to issue Pin-23,24 blocking/unblocking command.
11	Going to check pressing of <INS>, <END>key during power-on.
12	To init CMOS if "Init CMOS in every boot" is set or<END> key is pressed. Going to disable DMA and Interrupt controllers.
13	Video display is disabled and port-B is initialized. Chipset init about to begin.
14	8254 timer test about to start.
19	About to start memory refresh test.
1A	Memory Refresh line is toggling. Going to check 15us ON/OFF time
23	To read 8042 input port and disable Megakey GreenPC feature. Make BIOS code segment writeable.
24	To do any setup before Int vector init.
25	Interrupt vector initialization to begin. To clear password if necessary.
27	Any initialization before setting video mode to be done.

28	Going for monochrome mode and color mode setting.
2A	Different BUSes init (system, static, output devices) to start if present.
2B	To give control for any setup required before optional video ROM check.
2C	To look for optional video ROM and give control.
2D	To give control to do any processing after video ROM returns control.
2E	If EGA/VGA not found then do display memory R/W test.
2F	EGA/VGA not found. Display memory R/W test about to begin.

诊断码 (Hex)	所进行到的程序代码流程、停止或错误发生原因
30	Display memory R/W test passed. About to look for the retrace checking.
31	Display memory R/W test or retrace checking failed. To do alternate Display memory R/W test.
32	Alternate Display memory R/W test passed. To look for the alternate display retrace checking.
34	Video display checking over. Display mode to be set next.
37	Display mode set. Going to display the power on message.
38	Different BUSes init (input, IPL, general devices) to start if present.
39	Display different BUSes initialization error messages.
3A	New cursor position read and saved. To display the Hit <DEL> message.
40	To prepare the descriptor tables.
42	To enter in virtual mode for memory test.
43	To enable interrupts for diagnostics mode.
44	To initialize data to check memory wrap around at 0:0.
45	Data initialized. Going to check for memory wrap around at 0:0 and finding the total system memory size.

46	Memory wrap around test done. Memory size calculation over. About to go for writing patterns to test memory.
47	Pattern to be tested written in extended memory. Going to write patterns in base 640k memory.
48	Patterns written in base memory. Going to findout amount of memory below 1M memory.
49	Amount of memory below 1M found and verified. Going to findout amount of memory above 1M memory.
4B	Amount of memory above 1M found and verified. Check for soft reset and going to clear memory below 1M for soft reset. (If power on, go to check point# 4Eh) .
4C	Memory below 1M cleared (SOFT RESET) Going to clear memory above 1M.
4D	Memory above 1M cleared (SOFT RESET) Going to save the memory size. (Goto check point# 52h) .
4E	Memory test started (NOT SOFT RESET) About to display the first 64k memory size.
4F	Memory size display started. This will be updated during memory test. Going for sequential and random memory test.
50	Memory testing/initialization below 1M complete. Going to adjust displayed memory size for relocation/ shadow.
51	Memory size display adjusted due to relocation/shadow. Memory test above 1M to follow.
52	Memory testing/initialization above 1M complete. Going to save memory size information.
53	Memory size information is saved. CPU registers are saved. Going to enter in real mode.
54	Shutdown successful, CPU in real mode. Going to disable gate A20 line and disable parity/NMI.
57	A20 address line, parity/NMI disable successful. Going to adjust memory size depending on relocation/shadow.
58	Memory size adjusted for relocation/shadow. Going to clear Hit <DEL> message.

诊断码 (Hex)	所进行到的程序代码流程、停止或错误发生原因
59	Hit <DEL> message cleared. <WAIT...> message displayed. About to start DMA and interrupt controller test.
60	DMA page register test passed. To do DMA#1 base register test.
62	DMA#1 base register test passed. To do DMA#2 base register test.
65	DMA#2 base register test passed. To program DMA unit 1 and 2.
66	DMA unit 1 and 2 programming over. To initialize 8259 interrupt controller.
7F	Extended NMI sources enabling is in progress.
80	Keyboard test started. clearing output buffer, checking for stuck key, to issue keyboard reset command.
81	Keyboard reset error/stuck key found. To issue keyboard controller interface test command.
82	Keyboard controller interface test over. To write command byte and init circular buffer.
83	Command byte written, Global data init done. To check for lock-key.
84	Lock-key checking over. To check for memory size mismatch with CMOS.
85	Memory size check done. To display soft error and check for password or bypass setup.
86	Password checked. About to do programming before setup.
87	Programming before setup complete. To uncompress SETUP code and execute CMOS setup.
88	Returned from CMOS setup program and screen is cleared. About to do programming after setup.
89	Programming after setup complete. Going to display power on screen message.
8B	First screen message and <WAIT...> message displayed. PS/2 Mouse check and extended BIOS data area allocation to be done.

8C	Setup options programming after CMOS setup about to start.
8D	Going for hard disk controller reset.
8F	Hard disk controller reset done. Floppy setup to be done next.
91	Floppy setup complete. Hard disk setup to be done next.
95	Init of different BUSes optional ROMs from C800 to start.

诊断码 (Hex)	所进行到的程序代码流程、停止或错误发生原因
96	Going to do any init before C800 optional ROM control.
97	Any init before C800 optional ROM control is over. Optional ROM check and control will be done next.
98	Optional ROM control is done. About to give control to do any required processing after optional ROM returns control and enable external cache.
99	Any initialization required after optional ROM test over. Going to setup timer data area and printer base address.
9A	Return after setting timer and printer base address. Going to set the RS-232 base address.
9B	Returned after RS-232 base address. Going to do any initialization before Coprocessor test.
9C	Required initialization before Coprocessor is over. Going to initialize the Coprocessor next.
9D	Coprocesor initialized. Going to do any initialization after Coprocessor test.
9E	Initialization after Coprocessor test is complete. Going to check extended keyboard, keyboard ID and num-lock. Keyboard ID com mand to be
A2	Going to display any soft errors.
A3	Soft error display complete. Going to set keyboard typematic rate.
A4	Keyboard typematic rate set. To program memory wait states.

A5	Going to enable parity/NMI.
A7	NMI and parity enabled. Going to do any initialization required before giving control to optional ROM at E000.
A8	Initialization before E000 ROM control over. E000 ROM to get control next.
A9	Returned from E000 ROM control. Going to do any initialization required after E000 optional ROM control.
AA	Initialization after E000 optional ROM control is over. Going to display the system configuration.
AB	To build MP table if needed.
AC	To uncompress DMI data and execute DMI POST init.
B0	System configuration is displayed.
B1	Going to copy any code to specific area.
00	Copying of code to specific area done. Going to give control to INT-19 boot loader.

此处我们只是先根据AMI BIOS 原厂技术文献列举原文信息，在往后的章节中，我们还会再介绍一遍 Post/Debug Code，同时配合着源代码的浏览，相信能帮助读者掌握 BIOS 的精髓。

除了 port 80h 的简易 Debug 之外，系统研发人员也开发了所谓的 Remote Debug 遥控协同除错的功能，它是利用另一部计算机做为遥控端，以 RS-232C 接口通过类似 LL5 的连接线，连接到待除错端的计算机平台，被除错端安装特制的 Debug Card，控制被除错端的计算机 BIOS 以单步执行(single step) 的方式，一个指令接着一个指令地执行，或者设置断点，并且将各种寄存器、标志位，以及系统的内存、I/O port 状态回传到遥控端的计算机上。

另外，微软在最新的 PC2001 规范中，还定义了一个新的 Debug Port 规范，虽然详细规格必须去信索取，而且似乎还在修订阶段，据笔者粗略地查阅网站上的文献，大意是这个 Debug port 既有 COM1，COM2 的扩展，也具备了原先 COM1，COM2 所没有的硬件特性，如可以 I/O Port PnP (Plug & Play)、对外信号线减少并成 Serial 的特性，同时传输速率最少在 57600bps、可以到达 115200bps 甚至更高。

至于详细的 Debug Port 规范，读者可参阅下列网址：

www.microsoft.com/hwdev/NewPC/debugspec.htm

3-4 BIOS 所要掌握的系统资源

这里所谓的系统资源，我们定义有下列三项：I/O Port (I/O 端口地址)、IRQ 中断号码以及 DMA 号码。

I/O Port

x86 除了以地址线(address line)来驱动、存取内存地址之外，也有 I/O 地址线的设计，用来做主板其他系统组件或外围 I/O 装置的初始、组态、读写的用途；而以 I/O 地址线指定的接口，有时我们也以 I/O Port 来称呼。例如 Port 80h，就是指 I/O 地址信号电位状态为 0010000000b (=80h) 时，所指定、接触到的接口或装置。

x86 的 I/O 地址线只设计 16 条，因此最大的 I/O 端口号码数为 2^{16}=65536，从 0～0FFFFh，在传统的 ISA 接口，只使用到最前面 1024 个(0000～03FFh)，port 0400h 以上则为 PCI 接口与 EISA 接口所使用。

以下是笔者所搜集、整理，一个 x86 平台大致上所要用到的 I/O 接口的列表：

占用 I/O 的系统、外围接口名称	占用的I/O位置
8237 DMA Controller, DMA 控制器	00h～0Fh (第一组) C0h～DFh (第二组)
8259A PIC (Programmable Interrupt Controller) 可编程中断控制器	20h、21h (第一组) A0h、A1h (第二组)
8254 PIT (Programmable Interval Timer) 可编程定时器	40h～43h (XT/AT) 44h～47h (PS/2)
8042 KBC (KeyBoard Controller) 键盘控制器	60h～64h
PS/2 POS (Programmable Option Select) 可编程选项控制器 (92h 被使用于 A20)	90h～96h
Math Co-Processor,x87 Unit (数学辅助运算处理器)	F0～FFh
Secondary IDE(第二组IDE 接口)	170h～177h
Primary IDE(第一组 IDE 接口)	1F0h～1F7h
GAME I/O 摇杆接口	200h～201h
Sound Blaster/AD Lib(声霸卡或兼容声卡接口)	220h～22Fh
Plug and Play Configuration Register (PnP 组态寄存器)	279h、A79h
Serial Port4(第四组COM Port 接口、COM4)	2E8h～2EFh
Serial Port2 (第二组COM Port 接口、COM2)	2F8h～2FFh
MIDI Port (MIDI接口)	330h、331h
Parallel Printer Port (打印端口) 近代南桥芯片组或 SuperI/O 的 LPT1 默认值	378h～37Ah
MDA/MGA单色文字 / 图形显示卡接口	3B0h～3BFh
EGA/VGA彩色文字 / 图形接口	3C0h～3CFh
CGA CRT 寄存器、控制与调色板(Palette)寄存器	3D4h～3D9h
Serial Port3 (第三组 COM Port 接口、COM3)	3E8h～3EFh
Floppy Diskette (磁盘驱动器接口)	3F0h～3F7h

Enhanced IDE (EIDE 接口)	3F6h、3F7h
Serial Port1(第一组 COM Port 接口、COM1)	3F8h～3FFh
PCI Configuration Register/address (PCI 组态地址寄存器)	0CF8h
PCI Configuration Register/data(PCI 组态数据读写端口)	0CFCh

至于 EISA 扩充槽上的 I/O Port 地址指派法则，基本遵循下列公式，假设位于 EISA 的第 **n** 号扩充槽(n=1～0Fh)，则 EISA 所对映到的 I/O 地址为：

n000h～**n**0FFh 驱动该适配卡的 I/O 端口地址

n100h～**n**3FFh 对映到 0100h～03FFh 端口地址

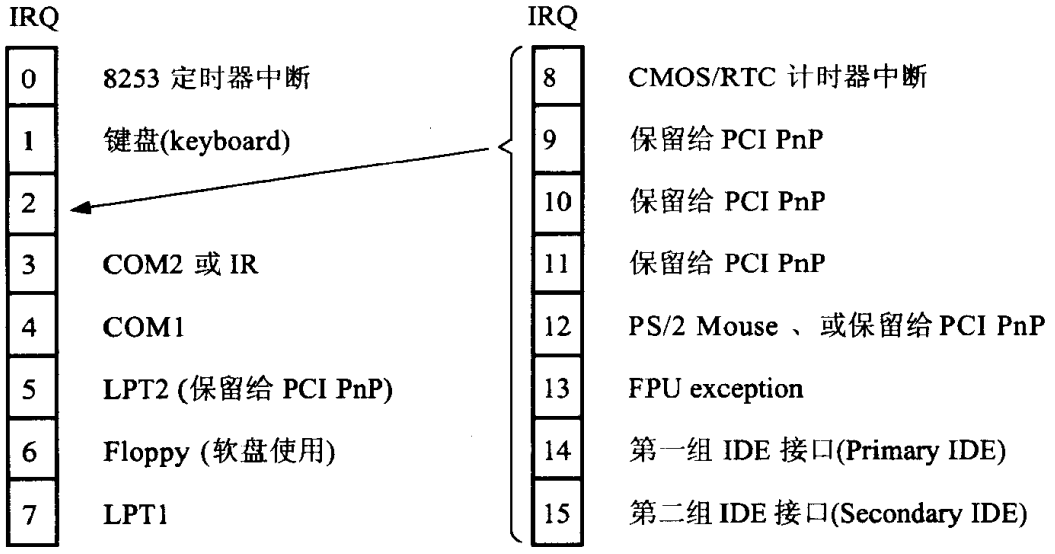
例如第 3 号 EISA 扩充槽，其占用地址为：

3000h～33FFh，其中 3000h～30FFh 为驱动第 3 号 EISA 扩充槽的 I/O 端口地址，而该卡若规划 3100h～33FFh，就会相对地涉及到主板上 0100h～03FFh 的 I/O 地址与相关的外围设备。

外围通知 CPU 处理的热线——IRQ (Interrupt ReQuest)

IRQ 是 Interrupt ReQuest (硬件中断要求) 的简称。一般说来 CPU 与外围设备相互联系的方式，除了通过 Memory，I/O 之外，还得借助 IRQ 的协助。当外围设备、适配卡或零组件有状况，极需 CPU 紧急处置(有数据输入／输出)的，该外围设备会产生一个硬件中断，然后通过主板上的 8259A 可程序化中断芯片 (PIC)，产生一组特定的 IRQ 信号，中断目前 CPU 的执行工作，首先根据该 IRQ 中断的顺序、号码，调用对映的中断处理程序先紧急处理后，再送出 EOI (End of Interrupt) 的信号，然后返回原先被中断的程序继续执行。

在 8086/88 (PC XT) 时代，只设计使用到一组 8259PIC，所以当时的 PC 只能处理八个 IRQ 中断，后来在 286 (AT) 推出时，就运用两个 8259PIC，以第二组 8259 的第二优先 IRQ，串接到第一组的 IRQ2，因而做出 16 组 IRQ 供外围设备使用，扩展至今成为 PC 的硬件中断结构：



当 CPU 接收到两个(或两个以上)的 IRQ 要求时，会依照它们的优先级，先处理 IRQ 等级比较高的中断需求，然后再去处理等级较低的需求。而它们的优先级是如下的：

← 高优先权处理　　最低优先权处理 →

0，1<8，9，10，11，12，13，14，15>3，4，5，6，7

因此我们可以发现，系统计时以及键盘按钮产生的中断，拥有至高无上的优先处理权，接下来是 CMOS/RTC 计时、PS/2 鼠标、第一组 IDE 设备、第二组 IDE 设备，COM2、COM1、软盘驱动器的处理排在倒数第二，LPT1 打印机的处理排在最后。

理论上比较密集而大量传输的外围设备，我们把它的 IRQ 优先权排在比较前面，它能获得 CPU 处理的机会也越大，对于传输效率或多或少会有所提升；但有些时候一个占用过多系统执行时间、IRQ 等级又占比较前面的设置，如果在 CPU 速度不够快，或芯片组的 PCI 安排设计不够 Smart 的情况下，有时会造成其他设备的无谓等待，连带的使得整个系统的运行效率下降。

今日 8259A 早已经内置在主板芯片组内，但是它的功用、规划方式并没有改变，16 组 IRQ 扣掉系统要用的，剩下四到六组可以给外围扩充使用；而将来 USB 装置越来越普及、Legacy Free 的概念普及之后，很有可能 COM2/COM1、LPT2、LPT1 与软盘驱动器接口都会逐渐淘汰，由 USB 1.1/2.0 来承担，届时 IRQ 能使用的就更多了。

当 PCI 总线加入主板之后，它又有一套全新的 IRQ 配置哲学：PCI 装置一律使用 INTA，INTB，INTC，INTD 四组 IRQ 中断讯号，这是随着 PCI 扩充槽或主板上的 PCI 装置(On Board PCI Device)的信号绕线法则来决定，在往后我们要介绍的 PCI IRQ Routing 章节时，会再仔细深入探究。

而事实上 INTA，INTB，INTC，INTD 信号线，最后都会通过 PnP BIOS 以及 Windows 的指派，从上述未使用的 IRQ 号码 (如 IRQ5，9，10，11，12) 选择并动态分配，而且在 PCI 规则中，PCI 装置的 IRQ 是可以共享的，当然它们的驱动程序也就得特别的设计了。

DMA (Direct Memory Access)直接存储器存取

DMA(Direct Memory Access)存储器直接存取技术，就定义上来说是一种外围组件直接掌控系统总线(Bus Mastering)来对存储器做数据传输／移动的操作，是一种加快设备传输效率的技术。传统设备需要数据传输时，发出 IRQ 中断通知 CPU，无论该设备是属于 I/O Mapped 或 Memory Mapped 的性质，CPU 都必须花费若干时间，从该设备对映的 I/O port 或存储器去读写数据，处理完毕后才能继续下一个工作。

运用 DMA 的设备，通常具有自己的控制芯片(如 SCSI 控制卡)，或者系统的南桥芯片支持，当 IRQ 中断信号产生时，CPU 只要针对该设备所指定的 DMA 信道做传输量的设置后，就由该设备直接取得内存总线的控制权，把数据写到存储器(或从存储器读出)；此时 CPU 的 I/O 处理时间缩短，特别是在多任务环境下，以往许多程序同时在执行时，存在因为某个程序正在做 I/O处理，所有的程序执行速度会骤然下降，大家都得停下来等这个程序处理完毕后才能继续执行的情况，DMA 技术可以使这种情况减到最低，多任务程序切换与执行更流畅。

目前运用 DMA 的设备，一开始软盘本身就有使用，随后声卡、SCSI 硬盘与部分网卡等也都有用到，最早期 IDE 硬盘还不支持 DMA 读写模式时，是通过 I/O Port 去读写扇区的，在多任务、密集 I/O 执行环境上比不上具备 DMA 传输功能的 SCSI 硬盘，而近年来 IDE 硬盘也加入了 DMA 传输模式(称为 Multi Word DMA)，搭配专门的驱动程序或新版的 Windows 98，Windows ME 或 Windows 2000等，也可以发挥出接近 SCSI 硬盘的流畅传输特点！

早期PC，XT (8088) 时代，PC 结构只具备一组 DMA 控制器 8237，共有四组 DMA 信道，每个信道宽度为 8bit 或 16bit，随着 AT 的出现，IBM 利用两组8237互相串联，目前 PC 结构的 8/16 位 DMA 信道一共有八组，列举如下：

通道号码	第一组8237 DMA 控制器 (8/16 位)
0	DRAM Refresh (DRAM刷新)
1	- - - - - - - - - - - - - - - - - -
2	软盘使用
3	开放使用

通道号码	第一组8237 DMA 控制器 (纯 16 位)
4	从 0～3 串接
5	开放使用
6	开放使用
7	开放使用

当近代芯片组纳入 PCI(Peripheral Component Interconnect)总线之后，PCI 规范中也加入了 32 位 DMA 的功能，而这部分大多是 IDE 硬盘或 PCI 外围设备所使用。目前由于 PCI 的 DMA 号码是采取动态配置，配合着 PnP BIOS 与 Windows 的协调，每组 PCI 装置都能够以独占(或分享 Share)的方式享有 IRQ 与 DMA 资源，这在 PCI 规范中是被允许的。

所以当你在近代 BIOS 的开机检测画面，看到好几组 PCI 设备共享一组 IRQ，DMA 时，请不要感到太过意外；只要 Windows 正确地识别出来，而且运行正常的话，就随它吧！

3-5 CMOS 的索引数据

前面已介绍过 CMOS RAM，我们都知道近代的 PC 结构，最少有 128bytes 的 CMOS RAM 的存储器，用来存放系统外围组态值，由 I/O port 70h 送出索引值 (bit6-0)，然后 Port 71h 就可以读写该索引值的字节数据。

至于 CMOS RAM 的索引值代表着什么意义呢？以下是它们的列表：

CMOS 的索引位置

索引位置	记录长度	数据域位的意义描述
00h-0Fh	16 bytes	计时芯片的日期/时间
10h-2Fh	32 bytes	传统 ISA 系统组态值
30h-3Fh	16 bytes	AMIBIOS 指派的系统组态值
40h-7Fh	64 bytes	扩展的 CMOS RAM，当今各芯片组均有实作出来。许多芯片组也由此存放特殊的组态信息

EISA 系统的 CMOS RAM 上限暴增到 8KB (=8192bytes)，而其中从 4,096 到 8,192 之间约 4KB 为非挥发性空间，被称为 EISA Extended CMOS RAM，而这部分也就是以后我们会介绍到的 General Purpose Non-Volatile RAM with I/O Mapped (以 I/O 端口对映的 GPNV 内存区域)。

这块区域一般是由 EISA Configuration UtilityECU) 组态程序所设定，也可通过 BIOS 的 INT 15h，AH=D8h Function 功能调用来读写 / 设定，而 EISA Extended CMOS RAM 也可以存取特殊 I/O 端口的方式进行读写 / 记录。

以下我们就针对这宝贵的 128bytes CMOS RAM，每一个索引地址代表的含义做具体的介绍：

CMOS RAM 的数据字段意义解释之一

偏移值 Offset	数据字段的意义描述 Description
00h	Real Time Clock (RTC) — 目前系统时间的“秒数”(Seconds) 字段
01h	Real Time Clock (RTC) — 预约警铃时间的“秒数”(Seconds) 字段
02h	Real Time Clock (RTC) — 目前系统时间的“分钟”(Minutes) 字段
03h	Real Time Clock (RTC) — 预约警铃时间的“分钟”(Minutes) 字段
04h	Real Time Clock (RTC) — 目前系统时间的“小时”(Hours) 字段
05h	Real Time Clock (RTC) — 预约警铃时间的“小时”(Hours) 字段
06h	Real Time Clock (RTC) — 星期几 (Sunday=01,Tuesday=02, etc)
07h	Real Time Clock (RTC) — 目前系统日期字段 (0～31)
08h	Real Time Clock (RTC) — 目前系统月份字段 (1～12)
09h	Real Time Clock (RTC) — 系统公元纪年的后两位 (00～99,00=2000, 01=2001, etc)
0Ah	Status Register A (状态寄存器 A) Bit 7 指示位。0= 目前可读取日期 / 时间，1= 日期 / 时间更新中，稍后再读取 Bits 6-4 计时频率除法器的填值，BIOS 默认为 010b，指定 32.768KHz 的计时频率 Bits 3-0 设置闹铃警示中断频率。BIOS 默认为 1024Hz，每次中断间隔为 1/1024 秒
0Bh	Status Register B (状态寄存器 A) Bit 7 停止系统频率设定时间(0=直接设定时间,实时系统仍在计时状态) (1=停止 RTC 计时并设定时间) Bit 6 周期性中断 (0=0=Disable，1=Enable) Bit 5 Alarm Interrupt (0=Disable，1=Enable) Bit 4 Update-Ended Interrupt (0=Disable，1=Enable) Bit 3 square wave 方波设定(0=关掉方波，1=依Status Reg A 设定产生方波) Bit 2 Date and Time Mode (0= 使用 BCD 格式，为默认值。1=binary 二进制值) Bit 1 24 或 12 小时计时制 (0= 指定 12 小时制，1= 设定 24 小时制) Bit 0 日光节约时间 (0=Disable，1=Enable)

0Ch	Status Register C(状态寄存器 A)，以下字节均为只读状态 Bit 7　IRQ Flag (read-only) Bit 6　Periodic Interrupt Flag (read-only) Bit 5　Alarm Interrupt Flag (read-only) Bit 4　Update Interrupt Flag (read only) Bit 3-0　保留未用，应该设为 0。
0Dh	Status Register D(状态寄存器 A) Bit 7　CMOS RAM 内容合法 (0=CMOS 电池储电量偏低，RAM 内容正常) (0=CMOS 电池储电量偏低，RAM 内容异常) Bit 6-0　保留未用，应该设为 0。
0Eh	Diagnostic Status(诊断状态记录值) Bit 7　CMOS/RTC 芯片电源 (0= 电源正常，1= 电源不正常) Bit 6　CMOS RAM checksum 加总检查值状态 (0=检查值正常，1=检查值不符) Bit 5　CMOS RAM 组态。0=CMOS 记录的组态与目前检测到的一致，1=组态不一致) Bit 4　CMOS RAM 记录的内存状态(0=CMOS 记录的内存跟目前所检测到的内存大小一致。1= 不一致) Bit 3　硬盘 C:启始状态(0= 启动通过，准备 boot。1= 硬盘 C:启动失败，无法boot) Bit 2　时间状态指示(0= 记录时间正常无误。1= 记录时间异常) Bit 1-0　保留未用，应该设为 0。
0Fh	shutdown status byte 当机复位指示字节，指示值如下： 00h　软件或其他不知名状态下的复位 (reset) 01h　正在真实 / 保护模式下检查内存时发生reset或者重新进入真实模式下做芯片组 initialization 时发生reset 02h　在真实 / 保护模式下内存检查通过后的系统复位 03h　在真实 / 保护模式下内存检查失败后的系统复位 04h　通过 INT 19h 重新开机 (boot) 05h　清除键盘中断(产生一个 EOI 信号)并且跳到 40h:0067h 记录的跳跃位置 06h　在保护模式下完成所有测试后的系统复位或在未产生EOI信号下跳到 40h:0067h 记录的跳跃位置 07h　在保护模式下没有通过所有测试后的系统复位 08h　由 POST 切到保护模式下进行内存检测时所使用 09h　由 BIOS INT1h 的 AH=87h (区块移动功能) 所使用

	0Ah	返回并跳跃到40:67h记录的程序进入点地址
	0Bh	以IRET方式回到40:67h记录的程序进入点
	0Ch	以RETF方式回到40:67h记录的程序进入点
	0Dh-FFh	系统电源刚打开时的复位

CMOS RAM 的数据字段意义解释之二

Offset	Description
00～10h	标准与IBM AT兼容的RTC计时芯片的时间/时间与状态寄存储器字段
11～13h	可供其他用途
14h	Bits 7-6 软盘驱动器数目 (Number of floppy drives) 00=1 Drive，01=2 Drives Bits 5-4 Monitor Type 显示器型号 00=Monochrome 01=40x25 CGA，10=80x25 CGA，11=VGA/EGA Bit 3 显示功能位 (0= 不显示，1= 启用并显示) Bit 2 键盘功能启用位 (0= 关掉键盘，1= 启用键盘) Bit 1 数学辅助处理器 (x87) (0= 不具备 FPU，1= 具备) Bit 0 软盘机 Drive (0=Disabled，1=Enabled)
15h	传统主存储器 KB 数，低字节。
16h	传统主存储器 KB 数，高字节 C0280h=640K
17h	与 IBM PC 兼容的总内存 KB 数，低字节
18h	与 IBM PC 兼容的总内存 KB 数，高字节 (最大值仅 FF FF=65535=64MB)
19h～2Dh	可供其他用途
2Eh	标准 CMOS RAM 加总检查值，低字节
2Fh	标准 CMOS RAM 加总检查值，低字节
30h	与 IBM PC 兼容的扩展内存 KB 数低字节
31h	与 IBM PC 兼容的扩展内存 KB 数高字节
32h	公元年号除以 100 后的 BCD 字段 (20h=2000～2099，跟 09h 字段凑成整个公元纪年)
33h～34h	保留未用

35h	POST 所检测到扩展内存 K 数，除以 64KB 后的数值 (低字节)
36h	POST 所检测到扩展内存 K 数，除以 64KB 后的数值 (高字节)
37h～3Dh	可供其他用途
3Eh	扩展 CMOS 加总检查值的低字节 (从 34h 计算到 3Dh)
3Fh	扩展 CMOS 加总检查值的高字节 (从 34h 计算到 3Dh)
40h～7Fh	供 AMI BIOS 动态配置使用

Interrupt Vector Table (中断向量表) 与号码

Intel 在规划 8086/8088 时，就设计了256个中断向量，分别从 INT 00h～INT FFh，一共是 256 组，每一组向量占用四个 bytes 。而真实模式下，中断向量表格的内存地址固定在 0000:0000～0000:03FFh，占用最前面 1KB 的内存空间。

以下是我们x86 结构计算机平台所使用到的中断向量表号码列表：

中断号码	属性	中断意义说明
INT 00h	(错)	CPU Divide Error (除法错误)
INT 01h	(硬)	CPU Single Setp(单步除错执行)
INT 02h	(硬)	NMI (Non Maskable Interrupt) 不可屏蔽中断
INT 03h	(硬)	Break point code (指令断点，指令码 0CCh)
INT 04h	(错)	CPU Overflow (数据溢出中断)
INT 05h	(软) (错)	BIOS 处理按下 Print Screen 键的打印服务程序接口 BOUND Range Exceed (寻址超越边界)
INT 06h	(错)	Invalid Op Code(非法 / 不支持的指令)
INT 07h	(错)	Floating Point Processor/FPU Not Avaiable (浮点运算处理器 / 浮点指令不支持)

INT 08h	(错)	IRQ0—System Timer (系统定时器中断) 第 8 号中断每 (1/18.2) 秒产生一次，用来做系统计时的工作
INT 09h	(硬) (错)	IRQ1 —Keyboard Interrupt (键盘中断) Processor Extension Protection ERROR 段选择扩展方向性错误
INT 0Ah	(硬) (错)	IRQ2 —串接到 IRQ8～15 的子程序 Invalid Task State Segment (TSS)，非法的工作任务区段错误
INT 0Bh	(硬) (错)	IRQ3 — COM2 (第二组串行接口) Segment not present (区段所对应的物理存储器不存在)
INT 0Ch	(硬) (错)	IRQ4 — COM1 (第一组串行接口) CPU Stack Fault (CPU 堆栈处理错误)
INT 0Dh	(硬) (错)	IRQ5 — LPT2(第二组打印端口)，现在保留给 PnP 系统配置 GENERAL PROTECTION VIOLATION(CPU (CPU 产生保护违规的错误)
INT 0Eh	(硬) (错)	IRQ6 — Floppy Diskette(软盘驱动器读写中断) PAGE FAULT 分页转换错误
INT 0Fh	(硬)	IRQ7 — LPT1(第一组打印端口)
INT 10h	(软)	屏幕I/O 接口,切换各文字 / 图形模式,提供显示 / 绘图卷页服务
INT 11h	(软)	PC 外围设备检查(PC Equipment)
INT 11h	(软)	数据 / 地址对齐检查错误，486 以上的 CPU 所产生
INT 12h	(软) (错)	传回 PC 主存储器大小检查 机器检查例外MACHINE CHECK EXCEPTION (Pentium以上的 CPU)
INT 13h	(软)	磁盘 I/O 接口 (软盘、硬盘读写、格式化等服务)
INT 14h	(软)	Serial Port Communication通讯端口中断服务例程
INT 15h	(软)	卡带接口服务程序，AT 扩展中断服务调用，程序多任务
INT 16h	(软)	键盘读取服务程序
INT 17h	(软)	打印服务程序
INT 18h	(软)	ROM BASIC 进入点地址，网卡 Diskless 开机拦截点
INT 19h	(软)	激活操作系统的进入点

INT 1Ah	(软)	BIOS 时间接口/CMOS RealTime Clock 电池时钟界面
INT 1Bh	(软)	BIOS的 CTRL BREAK 中断入点处理程序
INT 1Ch	(硬)	用户计时中断，从 INT8 处理子程序调用
INT 1Dh	(表)	视频适配卡参数地址表
INT 1Eh	(表)	软盘驱动器参数地址表
INT 1Fh	(表)	文字点阵字形表地址 (ASC80h～0FFh)
INT 20h	(软)	MS-DOS结束程序执行 (for .COM)
INT 21h	(软)	MS-DOS API (应用程序服务接口)
INT 22h	(软)	MS-DOS 程序结束跳跃地址 (进入点)
INT 23h	(软)	MS-DOS程序按下CTRL BREAK的中断进入点
INT 24h	(软)	MS-DOS Fatal Error Handler 严重错误处理程序
INT 25h	(软)	DOS Absolute Read (DOS 绝对扇区读取接口)
INT 26h	(软)	DOS Absolute Write (DOS 绝对扇区写入接口)
INT 27h	(软)	.COM 文件常驻存储器接口
INT 28h	(软)	DOS 闲置状态的系统调用 (应用在 TSR 常驻程序编写)
INT 29h	(软)	DOS Console Output (控制台输出)
INT 2Ah	(软)	网络软件接口层、NetBIOS
INT2Bh~2Dh	(软)	Reserved (保留给其他程序使用)
INT 2Eh	(软)	传递命令列参数给 DOS 命令解释器(SHELL=xxxxx.com)
INT 2Fh	(软) (软)	Multiplex Interrupt (多重系统中断 / 链接) CD-ROM 中断调用、HIMEM.SYS (XMS) 、Windows & DPMI Check

中断号码	属性	中断意义说明
INT 30h	(软)	Reserved (保留给其他程序使用)
INT 31h	(软)	DOS Protected Mode Interface (DPMI)，DOS 保护模式接口
INT 32h	(软)	Reserved (保留给其他程序使用)
INT 33h	(软)	MOUSE Service(鼠标中断服务程序)
INT 34h～3Eh	(软)	仿真浮点运算的仿真拦截码进入点
INT 3Fh	(软)	Overlay Manager
INT 40h	(软)	ROM磁盘中断服务程序
INT 41h	(表)	第一部硬盘参数地址表
INT 42h	(软)	旧的INT 10h 显示服务
INT 43h	(表)	EGA/MCGA/VGA 点阵字形数据表地址 (ASCII 00～7Fh)
INT 44h	(表)	PC jr 点阵字形数据表地址 (ASCII 00～7Fh)
INT 45h	(软)	Reserved (保留给其他程序使用)
INT 46h	(表)	第二部硬盘参数地址表
INT 47h～49h	(软)	Reserved (保留给其他程序使用)
INT 4Ah	(软)	CMOS/RTC Alarm Interrupt(定时器默认时间警示中断)
INT 4B～64h	(软)	保留未用
INT 65h	(软)	ADLib FM声卡调用服务程序
INT 66h	(软)	Reserved (保留给其他程序使用)
INT 67h	(软)	LIM EMS 3.2/4.0 服务
INT 68h～6Fh	(软)	Reserved (保留给其他程序使用)
INT 70h	(硬)	IRQ08 CMOS/RTC Timer interrupt，CMOS计时芯片中断
INT 71h	(硬)	IRQ09 重导向到INT 0Ah
INT 72h	(硬)	IRQ10 信号中断处理例程 (保留给 PnP 系统配置)
INT 73h	(硬)	IRQ11 信号中断处理例程 (保留给 PnP 系统配置)
INT 74h	(硬)	IRQ12 信号中断处理例程 (PS/2、USB...)
INT 75h	(硬)	IRQ13 信号中断处理例程(数学处理器错误中断)

INT 76h	(硬)	IRQ14 信号中断处理例程(for 第一组 IDE)
INT 77h	(硬)	IRQ15 信号中断处理例程(for 第二组 IDE)
INT 78h～7Fh	(软)	Reserved (保留给其他程序使用)
INT 80h～EFh	(软)	Reserved for BASIC 解释程序
INT F0h～FFh	(软)	Reserved (建议勿使用)

属性为

(软) 表示此由软件调用所产生，可提供服务并返回原调用的主程序

(硬) 表示由硬件 IRQ 信号产生，目前执行的程序必须先中断，处理完后继续执行

(错) 表示 CPU 检测到错误时中断目前的程序执行，处理后不一定返回原先断点。

(表) 表示此中断号码指向为一组表格位置，而非可调用、执行的程序代码进入点

CHAPTER

4

AMI源代码目录结构与文件维护

本书从现在开始，我们将真正开始介绍 AMI BIOS 比较深入的技术部分，而且是主板研发人员会接触到的部分。首先就从光盘上所附的 AMI BIOS 源代码文件目录开始探索起。

4-1 AMI BIOS 的源代码版本缘由

AMI 公司从 1985 年创立至今，BIOS 的研发超过十五年的历史，AT (286)，386，486， Pentium，P6(Pentium Ⅱ / Ⅲ，Celeron)到目前刚发表的 Pentium4，自然 BIOS 的源代码也经过若干版本的变动，年代比较久远(三年或 486 时期) 的就不多提了。

如果以笔者从 AMIBIOS98 Technical Reference、Porting Guide 这两本资料进行详述，应该是下列三种版本：

① version 6.26.10，简称 626.10。现已不采用。

② version 6.27.10，简称 627.10。

③ version 6.32，简称 632。美国 AMI 总部使用的版本。

最常使用的版本是 627.10 版本， 它是由 AMI 总部的 631 版本修改而来，除了加入了全新的 ACPI 功能模块之外，同时也将源代码目录设计成为了 CORE\CHIPSET\OEM 的层级结构。

美国 AMI 最新研发的 AMI BIOS Version 7.00(Core 7.0)，是全新结构的设计，有些公司则是引进并改成跟 627.10 类似、兼容的目录结构， 这个版本称为 7.00t，并且已经有少数项目在推广、使用中。

本书的介绍中将会以 627.10 的目录结构为主体。

4-2 AMI BIOS 的源代码目录设计

也许是模块设计上的需要，经过十几年变革的 AMI BIOS，已经逐渐定义了一个严谨有组织、有条理的磁盘目录设计原则，它将核心与次核心、零组件与附加模块之间的关系定义得十分严谨，BIOS 研发厂商或主板 BIOS 维护人员，只要习惯这种目录设计，日后要修改 BIOS，或接触新型主板的 BIOS 源代码时，可以很快地驾轻就熟，也有利于移交给底下新进人员或与同业人士讨论。

源代码文件最基本由下列的三个层次的目录所组成,如图4-1所示的结构：

① CORE(核心目录)：用来存储最深层核心的源代码与相关文件

② Chipset(芯片组目录)：用来存储跟特定系统芯片组有关的源代码与相关文件

③ OEM (OEM厂商目录) OEM系统大厂或主板厂商特定版本的区隔目录

其中 AMI Taiwan 最常使用的 627.10，它是将主核心目录摆在第一层，芯片组目录摆在第二层，而 OEM 厂商目录则摆在第三、第四层，其目录结构的顺序关系如下：

CORE\CHIPSET\OEM[\Customer]

例如以下是一个符合 627.10 的目录结构类型：

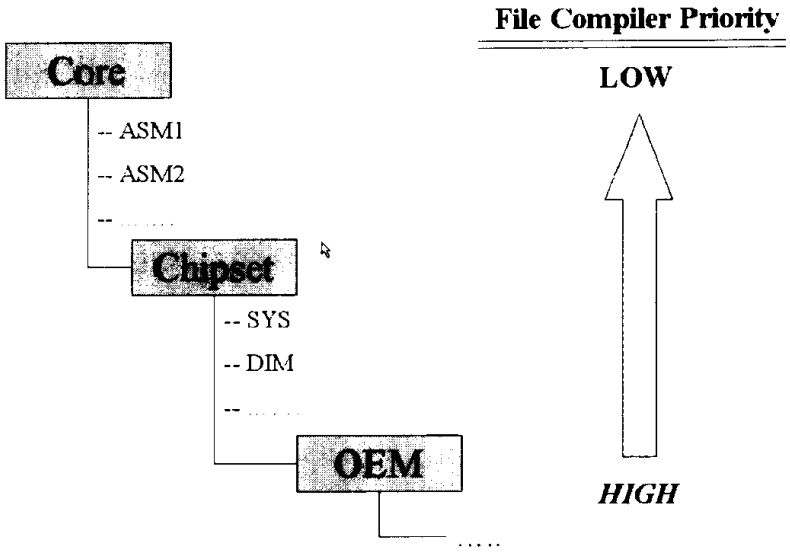
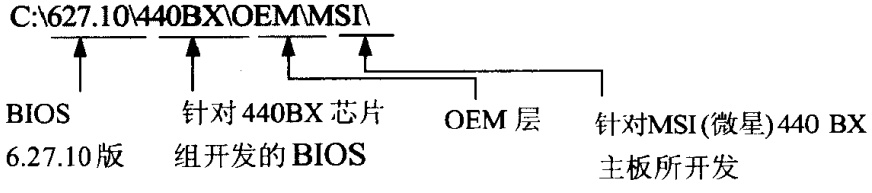


图 4-1　627.10 的目录结构与编译器处理的优先级



当然，随着主板客户对产品设计上便利性的要求，例如同样为 440BX 的主板，主板厂商可能设计出不同的特定版本，如使用某一家的 SuperI/O 芯片，某一家的音效芯片的特别版本等等，因此有时候会在从 OEM 底下再区分目录。

至于比较罕见的 631，632，则是将 OEM 厂商名称摆在最前面第一层，核心目录摆在第二层，芯片组目录摆在第三层；如果再有需要依 SuperI/O、音效芯片细分第四、第五层目录，其目录结构的顺序关系如图 4-2 所示：

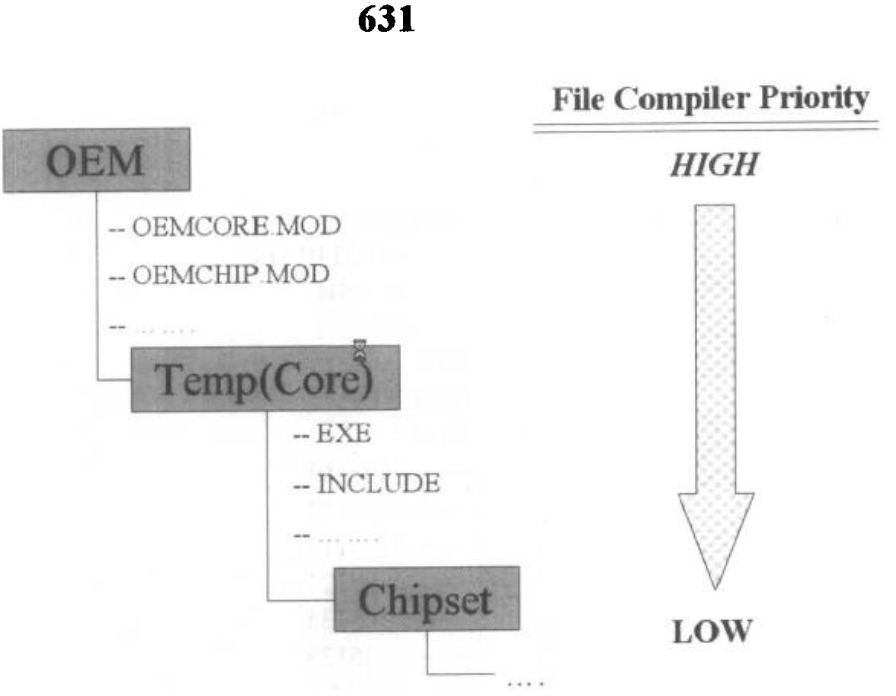
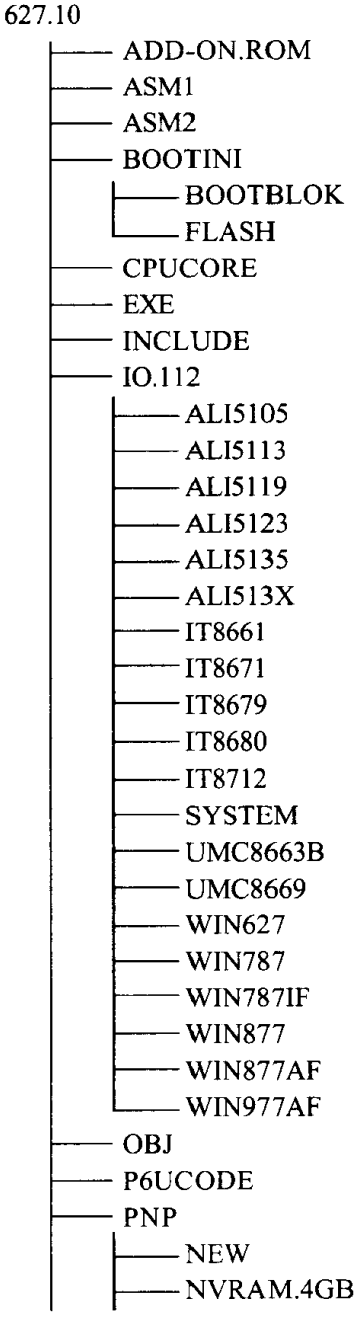
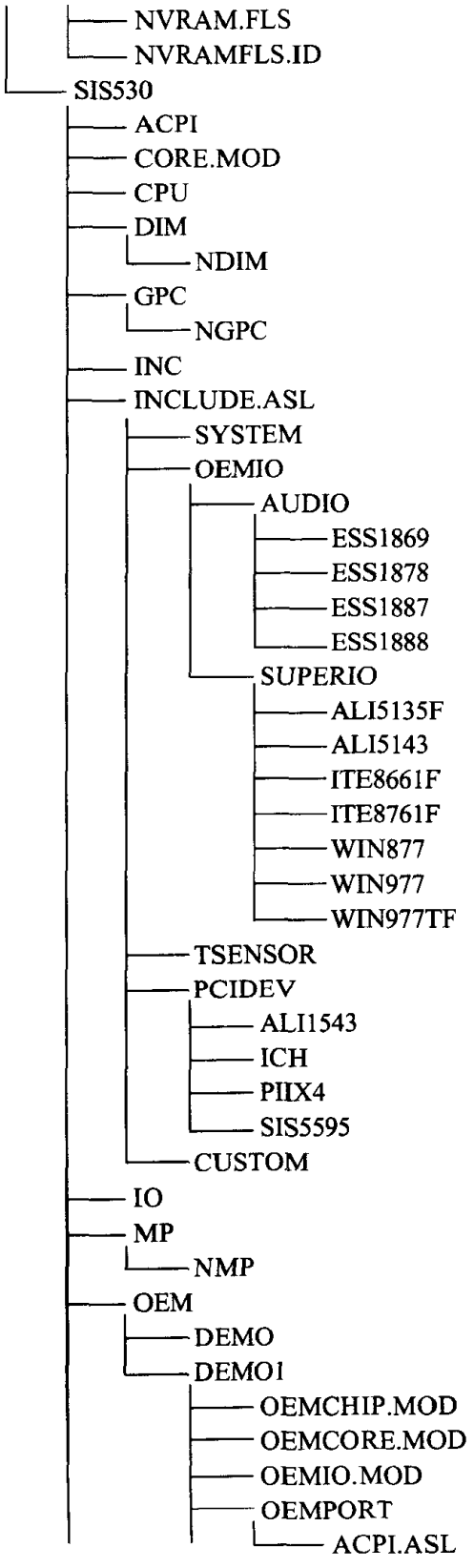


图 4-2　631 的目录结构与编译器处理的优先级

例如以下是一个符合 631 的目录结构类型：

由于627.10是开发者常接触到的版本，台湾AMI也授权这种目录结构的源代码烧写到光盘片中。因此，本书所要介绍的是以627.10的目录结构为主。以下是我们随书所附光盘中， AMI BIOS(627.10)以Sis530芯片组为结构下，整个源代码的目录树状结构图，如图4-3所示：




NVRAM.FLS
NVRAMFLS.ID
SIS530
ACPI
CORE.MOD
CPU
DIM
NDIM
GPC
NGPC
INC
INCLUDE.ASL
SYSTEM
OEMIO
AUDIO
ESS1869
ESS1878
ESS1887
ESS1888
SUPERIO
ALI5135F
ALI5143
ITE8661F
ITE8761F
WIN877
WIN977
WIN977TF
TSENSOR
PCIDEV
ALI1543
ICH
PIIX4
SIS5595
CUSTOM
IO
MP
NMP
OEM
DEMO
DEMO1
OEMCHIP.MOD
OEMCORE.MOD
OEMIO.MOD
OEMPORT
ACPI.ASL

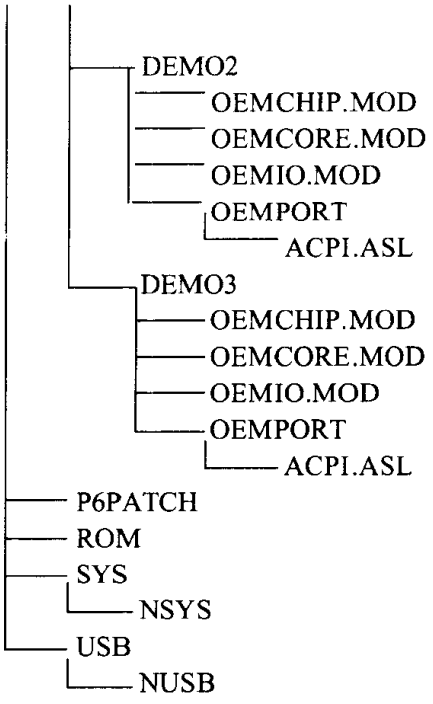


图 4-3　目录的树状结构图

主核心目录层(Core Directories)

\627.10　\ADD-ON.ROM ← 存放附加卡ROM文件的目录

\ASM1 ← 主核心源代码目录之一，包含 POST、Runtime 等

\ASM2 ← 主核心源代码目录之二，含 Setup 设置程序、CMOS 参数读写等

\BOOTINI ← BootBlock激活区块的程序代码、FlashPart烧写程序代码等

\CPUCORE ← 检测 CPU 品牌、型号与开关 Cache 、Register 等程序码

\EXE ← 跟源代码编译、链接与编修有关的工具程序存放的目录(进行编译作业时要将此目录加入 Path=的路径中)\

\INCLUDE ← 变量或宏定义引入文件(include files)

\IO.112 ← Super I/O 芯片程序代码文件，底下会再依 SuperI/O 芯片种类进一步细分

\OBJ ← 主核心编译出来的*.OBJ 文件会暂时存放在这个目录，供进一步处理

\P6UCODE ← 存放Pentium Pro以上的微码更新功能的源代码与微码文件

\PNP ← 跟Plug and Play(PnP)处理有关的子程序源代码

芯片组目录层(Chipset Directories)

\627.10\SIS530\

\ACPI ←跟此芯片组 ACPI 有关的程序代码存放目录

\CORE.MOD ←配合\627.10\ASM1、ASM2 等主核心要修改或附加的源文件存放目录

\CPU ←对应\627.10\CPUCORE 要配合、修改的源文件存放目录

\DIM ←Device Initialize Manager 装置初始 / 设置的源代码

\GPC ←跟绿色省电功能有关的程序代码目录

\INC ←芯片组变量、宏定义的引入文件

\INCLUDE.ASL ←ACPI 控制语言目录文件(底下还有更细分的目录)

\IO ←配合此芯片组外围 I/O 相关的源代码存放的目录

\MP ←配合此芯片组需要修改的多重处理器程序代码存放的目录

\P6PATCH ←配合此芯片组要修改的 P6 微码更新程序代码存放的目录(对应\627.10\P6UCODE)

\627.10\SIS530\

\ROM ←编译出来的.ROM 暂存盘

\SYS ←对应\627.10\ASM2, 配合此芯片组所要修改的设置画面、程序的相关程序代码存放目录

\USB ←USB 功能支持函数文件

OEM 目录层(OEM Directories)

\627.10\SIS530\OEM\DEMO ← 供读者自行练习移植的实作目录

\627.10\SIS530\OEM\DEMO1 ← 作者所实作的 Leadtek WinFast 5310MA 主板目录

\DEMO2 ← 本书作者所实作的 Soyo SY-5SSM 主板目录

\DEMO3 ← 本书作者所实作的技嘉 GA-5SSM 主板目录

OEM 厂商目录层(非必要，本书以DEMO目录下为例)

\627.10\SIS530\OEM\customer

\627.10\SIS530\OEM\DEMO\OEMCHIP.MOD ← OEM厂商BIOS高级选项(Advanced，Chipsets Power，PCIPNP.SSP 定义与相关程序代码目录，从芯片层\627.10\SYS530\SYS目录复制而来。

\OEMCORE.MOD ← OEM厂商BIOS主选项画面(CORE.SSP)、信息等源代码。从核心层\627.10\OBJ复制而来。

\OEMIO.MOD ← 厂商针对Super I/O芯片程序代码修改与相关画面选项区域(IOC、IO1/IO2/IO3.SSP)。从主核心\627.10\IO.112复制而来。

\OEMPORT ← 厂商针对ACP、POST、FlashPart、PCI Routing等修改的程序代码区。从核心层\627.10\BOOTINI与芯片层\627.10\SIS530\SYS等复制。

\ML ← MultiLanguage(多国语言目录文件，本书未提供)

\ML\CHINESE ← 中文字形、信息、相关修改源代码存放区

\ML\JAPANESE ← 日文字形、信息、相关修改源代码存放区

基本上你只要将整个627.10目录(连同底下的次目录与文件)，完整的复制到硬盘里面(例如C)，不管你是用DOS命令还是鼠标拖动来复制，总之我们的一切编译作业，都建议在Windows环境下，调出MS-DOS模式，然后用纯DOS环境下，使用自己最习惯的文本编辑程序(PE2，HE5，EDIT)来编写、编修程序代码就行了。

各位读者有没有想过，为什么要设计这样复杂的目录结构，而事实上很多芯片层甚至 OEM 层的目录、文件，都是由上一层(核心层或芯片组)原封不动整个复制过来，部分文件 / 目录再重新取名字。

因为 BIOS 的发展最重视“兼容性”，以及保留让模块随取、插入的原则。为了兼容性，我们对于最原始的核心层程序代码，一般原则是能不更动就不更动；要修改的话，是把它先从核心层复制到芯片层，然后去改芯片层的核心码(\627.10\ → \627.10\SIS530\)，而编译程序一旦察觉芯片层有较新日期、版本的源代码时，会优先编译芯片层的源代码并加以取代，而不会去编译、链接核心层的源代码，万一读者、研发者不慎改坏了，还有上一层核心层的最源代码可以依靠，再重头来过。

同理，OEM 厂商从 BIOS 或芯片组厂商接受了某个芯片组版本的 BIOS 源代码时，也是建议将核心层、芯片层等要更动的源代码，拷贝到更底下的 OEM 层目录，要修改则去改 OEM 层的源代码，通常 NMAKE 只是 Link 部分已 compiled 过的.obj 或.lib 而已。

而编译程序一旦察觉 OEM 层有较新日期、版本的源代码时，会将 OEM 层的源代码列为第一优先，芯片层目录的源代码列为第二优先，最后才是核心层目录的源代码。甚至连最下层重复定义了某个在上一层定义的变量时，也会以下层所定义的为参考值。例如芯片层的 BIOS.MAK，原本指定 BIOS_SIZE 长度为 1(128KB)，但到了 OEM 层发现不够，OEM.MAK 要将 BIOS_SIZE 这个变量改成 2(=256KB) 时，这样当 NMAKE.EXE 在执行维护工作时，就会以最下层的 OEM.MAK 的指定为优先，把 BIOS_SIZE 设定为 2(=256KB) 。

所以在本书的范例中，我们理所当然的要切入到 OEM 厂商层，去开发、维护 BIOS 源代码，除非必要，不然较上一层的芯片层甚至核心层，能不要改动就尽量不要改动。

编译 BIOS 最少所需要的目录从图 4-3 来看很明显，要编译一套最小的 BIOS，最少需要核心层(Core)的目录，而底下则是必备的目录文件：

627.10
- ADD-ON.ROM
- ASM1
- ASM2
- BOOTINI
  - BOOTBLOK
  - FLASH
- CPUCORE
- EXE
- INCLUDE
- OBJ
- SIS530(芯片层)
- CORE.MOD
- CPU
- DIM
  - NDIM
- INC
- IO
- ROM

如此编译出来的 BIOS，将不具备 ACPI，APM，MP(多处理器支持)，P6UCODE(P6 微码更新)等功能，等读者看完本书之后，可以自行尝试一下，能否造出一块文件长度微缩到仅 64KB～128KB 左右的超小型 BIOS。

至于每个目录底下的文件，由于数量实在太过庞大、琐碎，而有些是编译时所造出的临时参考文件，是可以随意改动的，因此，我们不会在此就每一个目录下的每一个文件都逐一说明。

但只要是跟编译、维护过程有关的工具程序，以及属于重点项目的操作维护文件(*.MAK)，甚至是 BIOS 源代码(*.ASM，*.EQU，*.INC) 等等，我们还是会在本章节以及往后章节(第 5～9 章)中详细介绍。

4-3 编译 BIOS 所需要的编译 / 链接程序

编译本书所附的源代码，需要下列编译 / 工具程序文件：

文件名称	建议版本	用途说明
NMAKE.EXE	1.40	程序开发维护工具(保护模式执行文件)
NMAKER.EXE	1.30	程序开发维护工具(真实模式执行文件)
MASM.EXE	6.11	Macro Assembler 6.11 主编译程序
ML.EXE	6.11	Macro Assembler 6.11 后勤编译程序兼链接程序(Linker)
LIB.EXE	3.20	Library Manager 链接库删减 / 编修工具程序
LINK.EXE	5.31	目的码(Obj) / 链接库连结器(Segmented Linker)

这些文件隶属于 Microsoft Assembler 6.11，或者在 Visual Studio 6.0、Visual C++、MSDN(Microsoft Development Network)等微软程序开发套件中找得到。为了维护智能财产权，这些编译程序请读者、研发人员自行备妥。

以 MASM 6.11 为例，它在硬盘内有下列树状目录结构：

```
MASM611
├──BIN          ←编译程式、工具程序可执行文件
├──HELP         ←帮助文件(非编译时的必要目录)
├──INCLUDE      ←线上变量、宏定义引入文件
├──INIT         ←在 Windows 环境下的设定文件目录
├──LIB          ←所附的子程序函数库(非必要)
├──SAMPLES      ←范例文件目录(非必要)
│   ├──DEMOS
│   ├──DOSDEV
│   ├──MIXED
│   ├──NTSAMPLE
│   │   ├──HELLO
│   │   ├──NTDLL
│   │   ├──SZSEARCH
│   │   └──THREADS
│   ├──PWBTUTOR
│   ├──SHOW
│   ├──TSR
│   ├──WINCLOCK
│   └──WINDLL
└──TMP          ←MASM611 所需要的临时工作目录(必要)
```

有粗体字形标示者代表此目录以及底下所有文件为 BIOS 编译时所必要的目录 / 文件

当读者将这些目录建立、文件安装妥当，记得在 AUTOEXEC.BAT 开机自动执行这些文件中，将 c:\masm611 以及 c:\masm\bin 加入搜寻路径 (PATH=...)，并加入下列命令叙述：

```
set include=c:\masm611\include\
set helpfiles=c:\masm611\help\*.hlp
set init=c:\masm611\init\
set asmex=c:\masm611\samples\
set tmp=c:\masm611\tmp
set demo=c:\masm611\demo
```

笔者也设计了一个简单的批处理文件 Setpath.bat, 放在 \627.10 目录下，读者只要执行过一次，就会设置好一切相关的路径；或者在 AUTOEXEC.BAT 加入 setpath.bat 的语句就更方便了。

ASL 的取得方式

ASL.EXE 是微软针对 ACPI 控制语言所开发的 ASL 与 AML 语言编译器。如果读者需要编译出一套具备 ACPI 控制功能的 BIOS 程序影像文件，则必须借助这个不附在本书光盘上的程序。当然你要是只想编译单纯 APM 功能的 BIOS，则不需要这组工具程序。

首先请先到下列网址(网页如图 4-4 所示) 。

http://www.microsoft.com/HWDEV/onnow/default.htm

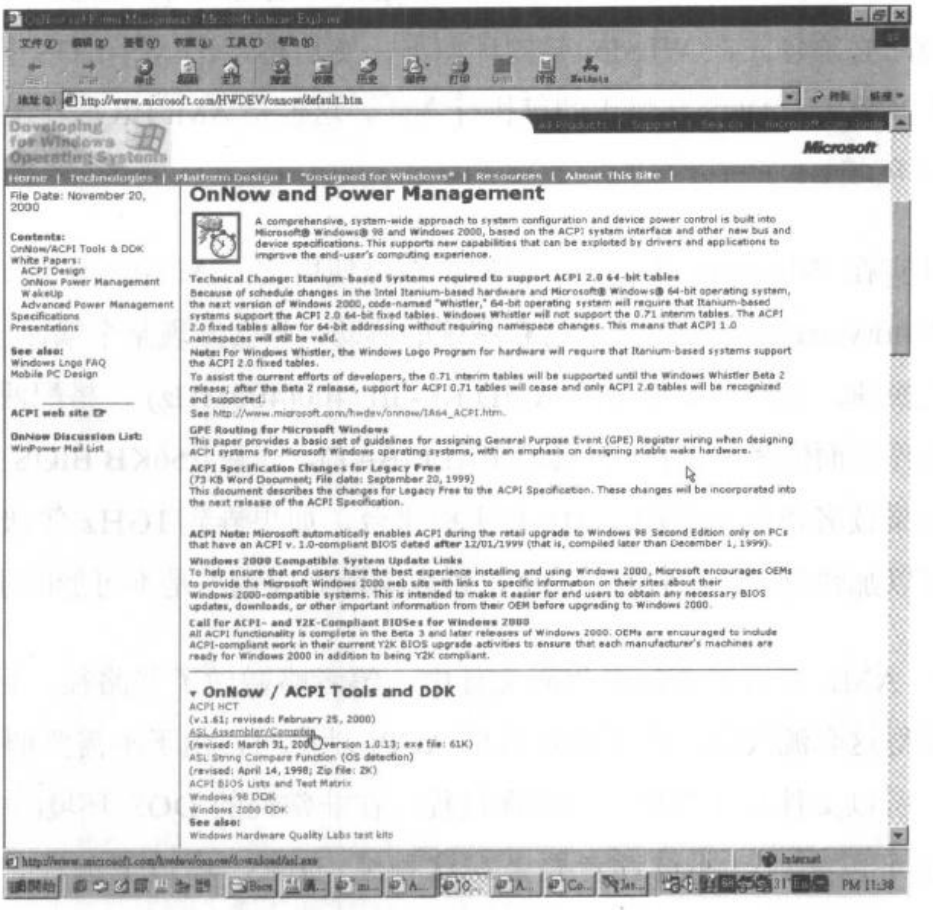


图 4-4 从网页中取得文件

取得文件之后，请把 ASL.EXE 复制到 C:\627.10\EXE 的目录底下。

至于有关于 ASL、ACPI 等技术部分以及相关章节，可参考第 2 章，或参考下列网址：

http://www.teleport.com/~acpi

编译的环境建议

由于 ASL.EXE 这个可执行文件，经过测试不可以在单纯的 DOS 开机模式下运行，非得在 Windows 环境下才能运行，这也使得我们整个 BIOS 的编译环境，必须建立在 Windows 的环境下，然后再调出 MS-DOS 模式来执行其他大多数属于 DOS 环境下的可执行文件。这也是 AMI BIOS 在 1998 年以后的建议环境。

其实在 Windows 环境下，我们仍然是使用它的 MS-DOS 模式，但由于有 SMartdrv.exe 这种磁盘文件寄存系统的协助，你会发现整个编译过程非常流畅而且快速。以笔者所使用的 AMD K6- III /400(400MHz)，搭配两块 SIS530 主板来作全面性编译工作(下 MAKEALL.BAT)，一块 256KB BIOS 仅 25 秒完成，如果读者以更高档的 CPU 与主板平台，如果弄到 1GHz 等级的 CPU，也许可以加快到 12～15 秒，甚至十秒内编译完成也不是不可能的事！

在 AMI 交给笔者的源代码文件中，笔者略微做了些路径、目录上的修改，使得这套源代码，在不指定启用 ACPI 功能时，由于不需要调用到 ASL.EXE，所以文件与工具程序的编译过程，在非常纯的 DOS 环境(如 MS-DOS 6.2)也可以完成。不过读者要是在这种纯 DOS 环境下编译，最好先在 CONFIG.SYS 挂入 HIMEM.SYS，同时在 AUTOEXEC.BAT 先加入下列命令：

```
SMARTDRV 4096 1024 A- B-  ←指定 4MB 磁盘寄存空间给磁盘驱动器 C:(软盘不需要)
```

当然在一般情况下，我们都会去编译具有 ACPI 功能的 BIOS 出来用，因此我们还是建议您在 Windows 环境下，选择 MS-DOS 模式后，再以较熟悉的 DOS 操作环境来编译 BIOS；而且最少是 Windows 98 以上的版本，包括 Windows 98SE 或 Windows ME 系统。

4-4 BIOS 的编译／维护作业的相关文件之一(IO.MAK)

以前在 DOS 时代，写过较大型汇编语言的读者，应该都知道维护文件(makefile)，它是用来叙述一套软件中的每个构成组件(含源代码 *.ASM、目标文件 *.OBJ、函数库 *.LIB 、执行文件 *.EXE、*.COM 或 *.BIN)，每个源代码、子程序的编译程序，以及链接的程序，都可以写成一个维护文件 (即 *.mak)，而且在编译、链接的过程中，维护的工具程序会考虑到每个组件的建立日期与时间的先后顺序，以“最新且有变动的部分才重新编译、连结”的原则，来进行整个源代码各成员的编译，最后连结、拼凑成一个完整的程序代码产品。

我们通过 MASM6.11 中的种种工具程序来帮助我们编译、连结出我们所要的 BIOS 监控程序，其中 NMAKE.EXE，NMAKER.EXE 就是帮我们维护整套 BIOS 的幕后功臣。当然，AMI 公司也写了几套语法复杂的 make file (即 *.MAK)，来帮助我们将这几十甚至几百个大大小小散落于各目录的文件，编译、连结并拼凑出我们(OEM 厂商)所期望的 BIOS 监控程序。下面我们就介绍这几组文件。

当我们将光盘中的源代码复制到硬盘，也设置好相关的环境变量(路径)之后，接下来请切换到 C:\627.10 目录，也就是最上层的核心层(Core layer)，我们会看到下列这些维护文件：

位于核心层(Core layer，\627.10 目录下的维护文件)

相对于 OEM 厂商层的路径 / 文件名	从根目录开始绝对路径 / 文件名	文件用途与操作说明
..\..\..\ASM1\ASM1.MAK	\627.10\ASM1\ASM1.MAK	开机 POST、常驻函数等相关源代码编译 / 维护文件
..\..\..\ASM1\CHINESE.MAK	\627.10\ASM1\CHINESE.MAK	多国语言模块的中文显示 / 切换源代码维护文件
..\..\..\ASM1\JMP.MAK	\627.10\ASM1\JMP.MAK	跳越表格源代码编译 / 维护
..\..\..\ASM2\ASM2.MAK	\627.10\ASM2\ASM2.MAK	关于 SETUP，CMOS 读写程序的编译 / 维护文件
..\..\..\BOOTINI\BOOT.MAK	\627.10\BOOTINI\BOOT.MAK	关于激活区块与FlashPart源代码编译 / 维护文件
..\..\..\PNP\PNP.MAK	\627.10\PNP\PNP.MAK	关于 PnP 与 ESCD 等子程序的编译 / 维护文件

切换到 C:\627.10\SIS530 芯片层，我们会看到下列这些 *.mak 文件：

位于芯片层(Chipset layer，\627.10\SIS530 目录下的维护文件)

相对于 OEM 厂商层的路径 / 文件名	从根目录开始绝对路径 / 文件名	文件用途与操作说明
..\..\BIOS.MAK	\627.10\SIS530\BIOS.MAK	关于 SIS530 芯片层的 BIOS 组译 / 维护文件
..\..\CORE.MAK	\627.10\SIS530\CORE.MAK	芯片层与核心层相关源代码编译、维护，以及将各模块连结成整块 BIOS 的工作
..\..\OBJ.MAK	\627.10\SIS530\OBJ.MAK	关于芯片层与核心层目标码(*.OBJ)的连结与维护文件
..\..\PATH.MAK	\627.10\SIS530\PATH.MAK	整个芯片与核心层的操作路径维护文件
..\..\DMI.MAK	\627.10\SIS530\DMI.MAK	关于 DMI(Desktop Manager Interface)功能模块的编译与连结的维护文件

前面也说过芯片层(\627，10\SIS530)甚至核心层的源代码的敏感程度，因此在这儿要强烈地建议读者，在没有熟读这些 MAK 文件内部的说明，或完全了解这些 MAK 文件的详细流程之前，不要轻易地改动内部的任何数值与字符串！否则“牵一发而动全身”，很可能造成日后 BIOS 无法顺利编译、维护，或其他不可预知的情形。

我们可以改动、实作的目录层应该是 OEM 厂商层，请切入到 C:\627.10\SIS530 \OEM\DEMO(或 DEMO1、DEMO2 等厂商层)。

我们会看到下列这些文件：

位于 OEM 厂商层(OEM vendor layer，\627.10\SIS530\OEM\xxxx\ 目录下的维护文件)

文件名称	用 途
IO.MAK	指定 Super I/O 芯片种类、I/O 索引值与读写端口等相关设置
OEM.MAK	指定 BIOS 大小、各种规格/功能引入、设定接口与附加卡程序代码等
ASLPORT.MAK	PCI IRQ Routing 所需要的设置维护文件，针对 ACPI 接口。
ASLBUILD.MAK	编译并建立 BIOS 所需要的 ASL 语言与相关表格
MAKEASL.MAK	编译并建立 BIOS 所需要的 ASL 语言与相关表格
MAKEBIOS.BAT	只编译/连结出有最新更动过的源代码(*.ASM) 并造出 BIOS 文件
MAKEALL.BAT	只要有一组最新更动的源代码，一切有必要的.ASM 与.OBJ、.LIB 源代码全部重新编译、连结与建立 BIOS 文件，效力最大

在这些文件中，我们在本章只介绍 IO.MAK，OEM.MAK，MAKEBIOS.BAT 与 MAKEALL.BAT 四个文件，因为这四组文件是可接触到 OEM 厂商层的厂商研发人员(或读者)，可以自行更改、修改的项目。

IO.MAK(控制 I/O 设置值的维护文件)

以 AMI BIOS 所附的源代码而言，开放给 OEM 层研发 / 维护人员接触的第一个维护文件，就是 IO.MAK。你必须在 MS-DOS 模式下，以 DOS 的文本编辑程序打开 OEM.MAK，然后依该主板所采用的 Super I/O 芯片的厂牌 / 型号，将对应的 SuperI/O 芯片字段设为 1；同时依照 IO.MAK 对各 Super I/O 芯片的说明，填入对应 ITE8661F 芯片的 IO_INDEX_PORT(索引端口值)、IO_DATA_PORT(数据读写端口值)。

有些 Super I/O 芯片还要填入 IO_UNLOCK_DATA(IO 数据解锁端口)以及 IO_ISA_PNP_DATA_PORT(ISA/PNP 数据读写端口)，在 IO.MAK 文件中已经有很清楚的说明。如此一来，当您修改完必要的源代码，执行 MAKEBIOS.BAT 或 MAKEALL.BAT 时，批处理文件就会自动调出 AMI 为该 SuperI/O 芯片所需要的并行端口、串行端口、红外线传输端口(IR Port)、软盘读写接口等相关子程序与目标文件(*.OBJ)，然后跟你的主程序连结。

如果在编译 BIOS 并写入 FlashEPROM 后重新开机，结果发现 Post Card 的数值停留在 2Ah 或 38h，这也是 OEM 厂商 BIOS 研发 / 维护人员经常遇到的问题。而会导致这两组 POST 码的可能原因有两个：不是 PCI IRQ Routing (PCI IRQ 绕线设置)的问题，就是 SuperI/O 这部分没有作好，读者可以从这两方面去寻求解决之道，顺利达成移植的目的。

本书所建议实作的两块 Sis530 芯片组为结构的主板(丽台 Leadtek WinFast 5310MA、梅捷 Soyo SY-5SSM)，都是采用编号 ITE8661F 这块 Super I/O 芯片，来处理 COM1、COM2、IR Port(红外线传输端口)、Printer Port(LPT1/EPP/ECP)等。我们切换到位于 OEM 层底下的厂商层(DEMO，DEMO1，DEMO2，...)，打开里面的 IO.MAK 来瞧瞧：

```
# ******************************************************************************
# ******************************************************************************
# **                                                                          **
# **                (C)Copyright 1985-1996, American Megatrends, Inc.         **
# **                                                                          **
# **                            All Rights Reserved.                          **
# **                                                                          **
# **                6145-F Northbelt Pkwy, Norcross, GA 30071                 **
# **                                                                          **
# **                            Phone (770)-246-8600                          **
# **                                                                          **
# ******************************************************************************
# ******************************************************************************
# ---------------------------------------------------------------------------- #
#                          SELECT IO MODULE                                    #
# ---------------------------------------------------------------------------- #
# NOTE TO ENGINEERS                                                            #
# 1. Select the IO chipset present in the motherboard by setting the           #
#    corresponding switch to 1. ←主板设计哪种 I/O 芯片就把该芯片标志设为 1     #
#    For example, if SMC665 is present in motherboard, set SMC665 = 1.         #
# 2. If onboard PCI IDE controller is present, please ADD the proper           #
#    routine in IO_PERIPHERALS in BIOS.MAK file. The corresponding             #
#    IOC.ASM and IOC.SSP file for PCI IDE is in CHIPSET\IO directory.          #
# 3. Generally you do not need to do anything else.                            #
# 4. But there are execptions. This make file is generated based on            #
#    following ASSUMPTIONs                                                     #
#    a) IDE support is NOT USED from any of these IO Chipsets.                 #
#       Nowadays IDE support is generally provided by onboard PCI IDE          #
#       chipset e.g. Intel Triton IDE (PIIX, PIIX3, etc.), SiS PCI IDE,        #
#       CMD PCI IDE, etc.                                                      #
#    b) All other devices (e.g. COM1, COM2, LPT, KBC, RTC, FDC) present        #
#       in the IO chipset are used in the motherboard.                         #
# 5. If any one of the above assumptions is NOT valid, you may need to         #
#    change/modify different switches according to the requirement.            #
#    For example:                                                              #
  a) If system is an EISA system                                               #
```

```
#         AND {(chipset does not have any PCI-EISA Bridge)                    #
#                  OR (PCI-EISA Bridge does not have proper class code)}      #
#      please add "SystemEisaBus" to IO_PERIPHERALS.                          #
#    b) If system is an ISA system                                            #
#         AND {(chipset does not have any PCI-ISA Bridge)                     #
#                  OR (PCI-ISA Bridge does not have proper class code)}       #
#      please add "SystemIsaBus" to IO_PERIPHERALS.                           #
#    c) If SMC932 is used but KBC is not used from SMC932, please TAKE        #
#      OUT the "Smc932BBlkKeyboardControllerAuto" from SMC932 block in        #
#      BOOT_KBC_RTC.                                                          #
#    d) If SMC932 is used but RTC is not used from SMC932, please TAKE        #
#      OUT the "Smc932BBlkRTCControllerAuto" from SMC932 block in             #
#      BOOT_KBC_RTC.                                                          #
#    e) If NAT306 is used and also IDE support is used from NAT306,           #
#      please ADD the "Nat306IdeController" in the NAT306 block in            #
#      IO_PERIPHERALS.                                                        #
# 6. The above examples are only a few of the many possible exceptions.       #
#    If your requirement is different from the above execptions, please       #
#    do it accordingly OR contact AMI.                                        #
# 7. If BIOS does not build with the selected IO support, please contact#
#    AMI for proper/latest IO library (IO.LIB, IO.PUB, IO*.SSP files).        #
# --------------------------------------------------------------------------- #
#         Select Index Port, Data Port, Unlock Value (if any)                 #
# --------------------------------------------------------------------------- #
# I/O Chipset          Possible Index     Possible Data     Possible Unlock Value
# ------------------   ------------------------------------ ---------------------
# AIP                  22/24/26E/398      23/25/26F/399
# ALI5105              Not used           Not used
# ALI5113              Not used           3F1/398
# ALI5119              Not used           Not used          19/29
# ALI5123              3F0/370            3F1/371
# CT711                3F0                3F1
# CT712                3F0                3F1
# CT735                3F0                3F1
# DS83CH20             2E                 2F
# GM803B               398                399
```

```
# GM803C               398              399
# ITE8661              3F0/3BD/370      3F1/3BE/371
# ITE8680              3F0/3BD/370      3F1/3BE/371
# Nat303               398/26E/15C/2E   399/26F/15D/2F
# Nat306               398/26E/15C/2E   399/26F/15D/2F
# Nat307               15C/2E           15D/2F
# Nat308               15C/2E           15D/2F
# Nat312               398/26E          399/26F
# Nat322               398/26E          399/26F
# Nat332               398/26E/15C/2E   399/26F/15D/2F
# Nat336               398/15C/2E/Any   399/15D/2F/(Any+1)
# Nat338               398/15C/2E/Any   399/15D/2F/(Any+1)
# SMC651               3F0              3F1
# SMC661               3F0              3F1
# SMC665               3F0              3F1
# SMC665IR             3F0              3F1
# SMC669               3F0/370          3F1/371
# SMC669FR             3F0/370          3F1/371
# SMC67X               3F0/370          3F1/371
# SMC68X               3F0/370          3F1/371
# SMC922               3F0/370          3F1/371
# SMC932               3F0/370          3F1/371
# SMC932FR             3F0/370          3F1/371
# SMC957               3F0/370          3F1/371
# UMC8663              108              109
# UMC8663B             108              109
# UMC8669 (Note2)      108              109
# Win787               251              252              88/89
# Win787IF             251              252              88/89
# Win877               3F0              3F1              86/87
#                      251              252              88/89
# NOTE1 : For Winbond 787(IF)/877, if the unlock data is 88h/89h then the# port
#         used to enable the access to the configuration registers is 250h, the index
#         port is 251h, the data port is 252h.
# NOTE2 : Also specify an I/O port for the ISA PnP RD_DATA port - In the
#         range of 203h - 3FFh
```

```
# ------------------------------------------------------------------------------#
IO_INDEX_PORT       = 3f0h #index port should be 3f0h (ITE8661F)          #
IO_DATA_PORT        = 3f1h #index port should be 3f1h (ITE8661F)          #
IO_UNLOCK_DATA      =          #e.g. 87h for Winbond877                   #
IO_ISA_PNP_DATA_PORT =         #RD_DATA Port e.g. 203h for UMC8669        #
# ------------------------------------------------------------------------ #
#              ONLY ONE IO SHOULD BE SELECTED                             #
# ------------------------------------------------------------------------ #
AIP          =0     #1=Include Intel AIP                                  #
ALI5105      =0     #1=Include ALI 5105                                   #
ALI5113      =0     #1=Include ALI 5113                                   #
ALI5119      =0     #1=Include ALI 5119                                   #
ALI5123      =0     #1=Include ALI 5123                                   #
CT735=0     #1=Include CT735                                              #
DSCH20       =0     #1=Include Dallas DS83CH20                            #
GM803B       =0     #1=Include LG Semicon GM82C803B                       #
GM803C       =0     #1=Include LG Semicon GM82C803C                       #
ITE8661      =1     #1=Include ITE 8661  ←指定使用ITE8661F SuperI/O 芯片#
ITE8680      =0     #1=Include ITE 8680                                   #
NAT303       =0     #1=Include National 303                               #
NAT306       =0     #1=Include National 306                               #
```

以下省略

```
WIN787_WIN777  =0   #1=Include Winbond 83787F/83777F                      #
WIN787IF       =0   #1=Include Winbond 83787IF                            #
WIN877         =0   #1=Include Winbond 83877                              #
WIN977         =0   #1=Include Winbond 83977                              #
# ------------------------------------------------------------------------ #
#              I/O CHIPSET SUPPORT IN BOOT BLOCK                          #
# ------------------------------------------------------------------------ #
# select the I/O peripheral for Boot Block code.                          #
# BOOT_KBC_RTC selects KBC and RTC routines controlled by I/O chipset.    #
```

```
BOOT_KBC_RTC  =                          ←指定由 SuperI/O 芯片控制键盘与 CMOS
!if$(ALI5123)                              计时器
BOOT_KBC_RTC  =$(BOOT_KBC_RTC)\
Ali5123BBlkRTCControllerAuto\
Ali5123BBlkKeyboardControllerAuto
!endif
```

以下省略

```
# BOOT_FDC    selects the FDC routines controlled by I/O chipset.          #
BOOT_FDC     =superiochip          ←指定通过 SuperI/O 芯片来读写软盘
(须该芯片也内置软盘接口才有意义)
!if$(AIP)
BOOT_FDC     =$(BOOT_FDC)\
AIPBBlkFloppyControllerAuto
!endif
```

以下省略

(本书所建议实作的两块以 Sis530 芯片组为结构的主板：丽台 Leadtek WinFast 5310MA 与梅捷 Soyo SY-5SSM，既然采用编号 ITE8661F 这块 Super I/O 芯片，所以笔者在 ITE8661 字段设成 1，其他项目一律设为 0。接下来按照 IO.MAK 的说明，查出 ITE8661F 芯片的 IO_INDEX_PORT=3F0h，IO_DATA_PORT=3F1h，也依照指示填入即可。

而 ITE8661 并没有 IO_UNLOCK_DATA 以及 IO_ISA_PNP_DATA_PORT 的项目，所以这两个项目填入空格符。

如果读者拿到的 Sis530 主板是其他种类的 SuperI/O 芯片，也可以依样画葫芦，查出对应的 SuperI/O 芯片并设置对应字段为 1，以及必要的 IO_INDEX_PORT，IO_DATA_PORT 即可，很简单吧！

若读者或研发人员希望编译出来的BIOS、控制键盘(Keyboard)以及CMOS计时器硬件部分，也改由SuperI/O芯片控制(这通常要取决于当初主板设计时的决定)，则可以在BOOT_KBC_RTC这个项目中，指定要赋予控制权的SuperI/O芯片的编号，下次开机时，就不再由传统的南桥芯片来控制，而是由SuperI/O芯片来控制。

最后，有些OEM/主板厂商，所采用到的南桥芯片与SuperI/O芯片，两者都具备软盘读写功能，造成功能上的重叠，此时系统默认值是采用南桥芯片来控制软盘，若希望软盘接口改由SuperI/O芯片控制，则可以在BOOT_FDC项目中，指定要赋予控制权的SuperI/O芯片的编号，编译好烧入FlashEPROM，下次开机，软盘驱动器就改通过SuperI/O芯片来控制啦。

4-5 BIOS的编译/维护工作的相关文件之二——供OEM厂商自定规格/功能等设置值的维护文件OEM.MAK

OEM.MAK是AMI BIOS开放给OEM/主板研发商可自行选择、添增/删减的重要维护文件。先介绍OEM.MAK文件中的一些字段与相关说明：

!INCLUDE *path\filename.mak*

定义：打开*path\filename.mak*的维护文件来操作，操作完成回到下一行继续执行。

例如!INCLUDE ..\..\bios.mak，则执行到这一行之前，会先调出上两层的bios.mak维护文件，也就是先去检查主核心层的源代码的异动情形，维护完毕再回到当前的OEM层继续作业。

REF_NUMBER = 009999

定义：指定 REF_NUMBER=009999，以供编译需要。

PROJECT_PATH=l*ccc*、PROJECT_BUILD=*nnn*

定义：指定 PROJECT_PATH=lchip，PROJECT_BUILD=nnn，最后生成 BIOS 程序影像文件名 BIOSFILE 为(PROJECT_PATH)+(PROJECT_BUILD).ROM=lchipnnn.ROM。

PROJECT_PATH 指定五位数字 lchip，第一位数字为 0 代表为美国亚特兰大(Atlanta)过来的源代码，1 则为台湾 AMI 所开发源代码。*chip* 代表该芯片组的四位数编号，例如 SiS530 以 SIS5 来取代。

PROJECT_BUILD 则指定三位数字 *nnn*，代表他们的第 *nnn* 号计划方案。该公司为 SIS530，所设置的计划代码为 300。最后生成的文件名为：

BIOSFILE=(PROJECT_PATH)+(PROJECT_BUILD).ROM=lSIS5300.ROM

如果读者想要更改编译出来的 BIOSFILE 文件名，例如想改成 MYSIS530.ROM，那么只要修改 BIOSFILE=MYSIS530.ROM 即可。不过笔者并没有去测试这样改动 OEM.MAK 的结果，会不会在编译／维护过程有不可预期的结果，笔者还是建议就按照它先生成 1SIS5300.ROM 文件，事后再 Rename 成您所要的文件就可以了。

NCPU=n

定义：指定目前主板是单处理器(NCPU=1)、双处理器(NCPU=2)或多处理器(NCPU=n)，SIS530 芯片组结构只支持单一处理器，所以这个字段只能设为 1，别无其他设置值。

若读者接触过双 CPU 主板，这个字段可设为 2 ，四个处理器的主板 / 系统，则设置为4。另外还需要跟 MPS_SUPPORT、EXTENDED_APIC 字段互相搭配。

BUP6=0|1

定义：指定是否启用 P6 特殊功能支持(P6 Support)，非 Intel P6 或兼容处理器芯片组一律设为 0 。

说明：这部分须配合另一个 FLASH_4GB 的字段设为 1，才能有效地完成编译作业。

MPS_SUPPORT=0

定义：主板是否具备多重处理器结构所需的 IO APIC 芯片或相关电路。(1= 有，0= 没有，0 为建议值)

这个字段是针对多重处理器的主板所需要的IO APIC 芯片或相关电路，加载并激活多重处理器的相关索引表格、寄存器等设置的子程序连结作业。还需要跟 NCPU，EXTENDED_APIC 字段互相搭配。

由于 SIS530 芯片组结构只支持单一处理器，无法外搭 IO APIC 芯片，所以这个字段只能设为 1，别无其他设置值；若读者接触过双 CPU 主板，并且在为它移植 BIOS，则这个字段可设为 2。同理，四个处理器的主板 / 系统，则设置为 4 。

目前仅 Intel 440BX、450NX、440GX 等芯片组，以及威盛的 694X 芯片组，可以外接 82093AA(IO APIC)芯片，供 OEM 或主板厂商设计双 CPU 的主板。

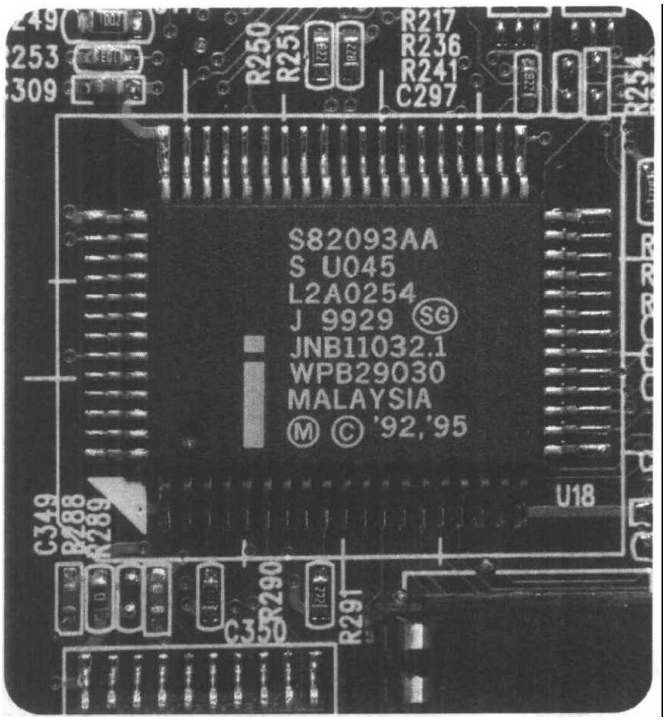


图4-5 常在 Intel 双 CPU 芯片组看到的 82093AA(IO APIC)芯片

EXTENDED_APIC = 0|1

定义：是否激活扩展的 APIC 中断管理功能(1= 多出 IRQ16～IRQ23 (IRQ23/SMI))

当激活 Extend_APIC=1 时，系统会多出 IRQ16～IRQ23 这八个 IRQ 中断信号，并且分配给 PCI 扩充槽去配置 IRQ 中断号码，其中 IRQ23 是 SMI (System Management Interrupt)给系统进入节电管理模式时所使用。

这个选项仅 MPS_SUPPORT=1 时才会有作用，通常在多处理器系统的主板，以及某些需要多一点的 IRQ(作为外围监控)的笔记本计算机系统，会需要启用这个选项，但是也需要有相关硬件线路、芯片的配合。

OEM_ID = *oemidstring*　PRODUCT_ID = *idstring*

定义：定义 OEM ID 以及 Product ID 的字符串。OEM ID 最多八个字符，而 Product ID 最多为 12 个字符。

例如：

```
OEM_ID     = SiS        <-------- 指定 OEM_ID 为 SIS
PRODUCT_ID = SiS530     <-------- 指定 Product_ID 为 SiS530
```

BIOS_SIZE= n(n=0～3)

定义：指定编译 / 连结后的 BIOS 监控程序影像文件大小。n=0 为 64KB，n=1 为 128KB，n=2 为 256KB，n=3 为 512KB 。

从 1998 年以后所出现的 BIOS，大多是 256KB 的长度大小，而最近 Intel 芯片组为结构的主板，则 BIOS 大小几乎都在 512KB 左右。

SOFT_OFF=0|1

定义：设置是否启用 ATX 软件电源关机功能(1=Yes, 0=No)。1 为建议值。

ACPI_SUPPORT = 0|1

定义：设置是否启用 ACPI 进阶组态电源管理接口的功能。(1=enable, 0=disable)

当 ACPI_SUPPORT 设为 1 时，此时 ATX 软件关机功能不再通过 BIOS 的 APM 协议(或以前的 GreenPC)，而是通过 ACPI 协议，由 OS(如 Windows 98，Windows ME)代为关闭。

ONCHIP_LM78 = 0|1 LM78_BASE_ADDRESS = *nnnh*

定义：设置主板是否内置 LM78(外围电压监控)芯片 / 电路(1= 有内置，0= 没内置)。以及指定 LM78 的 I/O 索引地址。

例 如：

```
ONCHIP_LM78      = 1          ←——主板设计了 LM78(或兼容)电压监控芯片
LM78_BASE_ADDRESS = 295h      ←—LM78 的起始 I/O 端口地址为 295h
```

LM75_SUPPORT = 0|1 LM75_SLAVE_ADDRESS = nnh

定义：设置主板是否内置 LM75(温度 / 风扇马达转速监控)芯片 / 电路(1= 有内置，0= 没内置) 。以及指定 LM75 的 I/O 索引地址。

例 如：

```
LM75_SUPPORT     = 1          ←——主板设计了 LM78(或兼容)电压监控芯片
LM75_SLAVE_ADDRESS = 9eh      ←—LM78 的起始 I/O 端口地址为 295h
```

SILENT_BOOT = 0|1

定义：激活 / 关闭全屏幕画面的显示功能(1= 启用开机全屏幕画面显示功能，0= 传统 POST 信息条列式开机)。

各位应该在许多笔记本电脑上，看过计算机开机时，并没有像台式计算机那样显示 POST 过程的条列式信息，而是显示出代表厂商 / 产品商标的全屏幕画面，之后才切换到 Windows 或其他操作系统的画面。AMI BIOS 也提供这样的设计，而这种隐蔽 POST 信息，以全屏幕画面显示的开机模式，就称为 Silent Boot 。

在编译的过程中，如果厂商指定了 SILENT_BOOT=1，接下来要指定 OEM_LOGO_FILENAME 这个字段，把要显示的全屏幕图像文件数据引入，并压缩成为 BIOS 内部的一个模块。详细细节，我们会在第 10 章硬件移植实战 DIY 中再次说明。

- **GPNV_SUPPORT = 0|1**

定义：指定启用 **GPNV(General Purpose Non-Volatile)**内存读写区域(1=enable)。

- **GPNV_FLASH = 0|1**

定义：指定以 FlashEPROM 的 NVRAM 区域作为 GPNV(=1)，或者以 CMOS RAM 为 GPNV(=0) 。

GPNV 在字义上是一块供 BIOS 存储组态设置、参数数据，而且不需担心关掉电源后数据会消失的内存区域(即非挥发性内存)，目前 CMOS RAM 以及 FlashEPROM 内存都具备这种特性；FlashEPROM 还特地设计一个区块就叫 NVRAM 区域，供 BIOS 作为 GPNV 之用。

以 I/O 端口来指定、读写的 GPNV，称为 IO Mapped NVRAM，如有些早期的 32bit EISA 总线系统，它们的 CMOS RAM 多达 8KB～16KB，因此可以借助 CMOS RAM 来作组态设置、记录参数之用，读者若是移植这类平台的 BIOS，就可以指定 GPNV_FLASH=0，以 IO Mapped NVRAM 作为 GPNV。

但是一般 Wintel(x86) 结构的系统平台，CMOS RAM 只有 128bytes，加上一些从 AT 时代保留至今的字段，事实上能使用的组态记录值可能不超过七、八十个字节，一般只能用来存储一些设备检测、存储器容量记录以及外围设备的开启 / 关闭。

至于像 Desktop Management Interface(DMI) 以及后来 SMBus BIOS 要记录的厂商文件，包含 CPU、Chipsets 、主板各零组件容量、设置值等字符串，有时候多达 20KB，这么多的文件根本没办法存放到 CMOS RAM，就得借助容量较大的 FlashEPROM 存储才行。

因此，只要你想编译的 BIOS 需要 DMI 功能，一般的情况下必须将 GPNV_SUPPORT 设为 1，同时 GPNV_FLASH 也设为 1，才能编译出正常运作且支持 DMI、SMBus 的 BIOS 程序代码。

EVENT_LOG = 0|1

定义：设置是否激活苏醒事件记录(Event Log，详情请看 ACPI 相关规范)的功能。

一般台式计算机的 BIOS 都默认为 =0，仅笔记本计算机才设置为 EVENT_LOG=1 。

USB_SUPPORT = 0|1

定义：设置是否开启 USB 设备的驱动能力(1=enable, 0=disable)。建议默认值为 1 。

SUI_FILE=*setupfil.bin*

定义：指定 Setup 设置接口程序形式的文件名称。

例 如：

```
SUI_FILE = SP130.BIN  ◄— 指定使用 Simple Setup V1.30 的设置接口。
```

SIGN_ON_MSG1="*string*",0

定义：指定在 POST 开机画面中，所显示的签名文件文字信息。结束字符以 00(ASCIIZ)代表。

例如：

```
SIGN_ON_MSG1  ='SIS-530 BIOS Version 1.0',0
```

开机时，只要是标准的 POST 画面(非 Silent Boot)，则会多出一行 SIS-530 BIOS Version 1.0 的字符串。

INTEL、SST、ATMEL、WINBOND、AMD

定义：指定这块主板所支持的 FlashEPROM 厂商种类。

例如：

```
INTEL       =0
SST         =1
ATMEL       =1
WINBOND     =1
AMD         =1
```

代表当前这块主板支持的 FlashEPROM 芯片形态，支持 SST、ATMEL、Winbond 以及 AMD 四家，但不支持 Intel 的 FlashEPROM 芯片烧写形态。

SMB_BASE_HIGH,SMB_BASE_LOW

定义：指定 SMBus 控制器的 Base Address 高字节与低字节。

默认值为 SMB_BASE_HIGH=004h，SMB_BASE_LOW=040h，请按照默认值设置，不要任意改动。

PM_BASE_HIGH、PM_BASE_LOW

定义：指定 Power Management Controller(ACPI)控制器 Base Address 高字节与低字节。

默认值为 PM_BASE_HIGH=004h，PM_BASE_LOW=000h，请按照默认值设置，不要任意改动。

OEM_LOGO_FILENAME=*filename.pcx*

定义：指定 BIOS 以 filename.pcx 图形文件嵌入，作为 Silent Boot 开机所显示的全屏幕图形。

要加载全屏幕显示画面的图形文件，长宽分别为水平 640 点、垂直 480 点(640x480)，256 色模式，而图形文件格式为.pcx。

受限于 BIOS 模块格式的定义限制，以及 FlashEPROM 的成本，AMI 对这个全屏幕图形文件的长度必须有所限制，据笔者的测试结果，最好是文件总长度小于 64000bytes(约 63KB 上下)，否则会有无法正常显示的状况发生。

ML_MODULE=0|1

定义：

激活多重语言支持功能(1=enable, 0=disable，只有英文单字节字集)。

这个字段是用来激活多重语言模块(Multi-Language Module) 的切换功能。

OEM 研发者必须先在 OEM 厂商层目录下生成一个 ML 的目录，然后在该目录下分别生成日文(JAPANESE)、韩文(KOREAN)、繁体中文(CHINESE)、简体中文等目录，然后再将 OEM 厂商层甚至更上一层芯片层的必要源代码，分别复制到各种不同语言的目录；然后你必须用支持该国语言的文本编辑器，先把一些重要的字符串信息给“当地语言化(Localize)”，保存到对应该语言文字的目录。

接下来借助工具程序的协助，将这些多出来的文字信息另外生成一个模块文件，然后在编译时，一起连结到 BIOS 内部。以下的 ML_MODULE_FILE，ML_MODULE_DESTSEG 以及 ML_MODULE_DESTADDR 字段，就是指定这个外挂的其他语言字符串的模块文件名、加载的内存段地址以及偏移地址。

ML_MODULE_FILENAME=*mlmodule.bin*

定义：指定这个外挂语言 / 文字附加模块的文件名称。

ML_MODULE_DESTSEG=*ssss*、ML_MODULE_ DESTADDR=*oooo*

定义：指定这个外挂模块被解压缩并加载执行的内存段地址(Segment) 为 *ssss*, 偏移地址(offest)为 *oooo* ，内存地址为 *ssss:oooo* 。

例如：

```
ML_MODULE_DESTSEG =7000h
ML_MODULE_DESTADDR=0000h
```

表示这个语言文字模块，会被 BIOS 解压缩到 7000：0000 这个内存地址。

关于多语言 文字附加模块功能的进一步说明

据 AMI 台湾分公司告知，目前 AMI BIOS 的中文、日文等多国语言的支持结构，以及整个外挂的模块设计与编译流程，甚至从 BIOS 内部模块，到工具程序都会有重大改变。目前多国语言的支持部分，AMI 公司并没有开放给 OEM 系统厂商 / 主板厂商来自行追加，而是以专案处理的方式进行，对于中文或其他国家语言 / 文字显示有特别需要的厂商 BIOS 研发人员，请从 AMI 台湾分公司寻求技术上的帮助。

PCI_ROMn=0|1

定义：是否引入第 n 组 PCI 附加卡的 ROM 影像文件(1=include, 0=not include)。当 OEM/ 主板研发厂商，将一些原本是属于 PCI 附加卡的电路功能，给整合到主板内，如果该附加卡有专属的 ROM BIOS 文件，此时可以通过 PCI_ROM*n* 这个字段，以及下面要介绍的 PCI_ROM*n*_FILENAME，PCI_ROM1_DEVICEID，PCI_ROM1_VENDORID，把这个附加卡的 ROM BIOS 程序代码嵌入整合到当前的 BIOS，成为系统 BIOS 的一部分。

例如大家所知道的 SCSI 附加卡，原本 SCSI 卡有自己的 SCSI BIOS，译码内存地址由该卡决定，把这张卡插到主板时，主板上就具备了 SCSI BIOS 以及本身的 MainboardBIOS 芯片，并且各自运作。但是当主板厂商想把该 SCSI 附加卡做进主板内时，除了在板子上多加该 SCSI 控制芯片以及相关电路之外，并不需要多做一个 EPROM 或 FlashEPROM 给 SCSI 控制器的程序代码存储。

只要原先的 Mainboard BIOS 还有空间，就可以把原先该型 SCSI 卡的 ROM BIOS 转成一个 ROM 文件，然后在编译时包含进来就可以了，不需要另外再设计一个 FlashEPROM 给该 SCSI 芯片的程序代码存储。

当主板上的 SCSI 芯片运行时，是执行被嵌入于主系统 BIOS 内部的 SCSI BIOS 程序模块，效果就跟原先插 SCSI 附加卡，由 SCSI 附加卡上的 FlashEPROM 内部的 SCSI BIOS 程序代码执行时，运行情况是完全相同的。

PCI_ROMn_FILENAME=*addonpci.rom*

定义：指定引入 *addonpci.rom* 这个附加卡影像文件。

PCI_ROMn_DEVICEID=*dddd*

定义：指定这个引入的附加卡影像文件中，附加卡的 PCI 装置 ID (Device ID)为 *dddd* ，单位为十六进制(Hex) 。

PCI_ROMn_VENDORID=*vvvv*

定义：指定这个引入的附加卡影像文件中，附加卡的 PCI 厂商 ID(Vendor ID)为 *vvvv* ，单位为十六进制(Hex) 。

以上 PCI_ROMn，PCI_ROMn_FILENAME，PCI_ROMn_DEVICEID 以及 PCI_ROMn_VENDORID 等字段，都是用来给 OEM/ 主板厂商添加外围芯片的程序代码之用。n 从 1～4 有四组，以下是一个取自 oem.mak 的字段设置范例：

指定引入第一组附加卡程序代码模块

```
PCI_ROM1          = 1
PCI_ROM1_FILENAME = 530v107.rom
PCI_ROM1_DEVICEID = 6306
PCI_ROM1_VENDORID = 1039
```

以当前目录下的 530v107.rom 为引入的程序代码模块文件。

引入的程序代码所控制的 PCI 装置代号为 6306(即 SiS530 的显示芯片电路的代号)

引入的程序代码所控制的 PCI 装置厂商代号 1039(1039h 为硅统科技在 PCI 联盟注册的代码)

USER_ROMn_FILENAME=*user_use.rom*

定义：指定引入 *user_use.rom* 这个使用者专属的影像文件。

USER_ROMn_DEVICEID=*dddd*

定义：指定这个引入的使用者自定的附加模块中，所控制的 PCI 装置 ID(Device ID)为 *dddd*，单位为十六进制(Hex)。

USER_ROMn_VENDORID=*vvvv*

定义：指定这个引入的使用者自定的附加模块中，所控制的 PCI 厂商 ID(Vendor ID)为 *vvvv*，单位为十六进制(Hex)。

USER_ROMn_DESTSEG=*ssss*，USER_ROMn_DESTADDR=*oooo*

定义：指定这个使用者自定的外挂模块，被解压缩并加载执行的段地址(Segment)为 *ssss*，偏移地址(offest)为 *oooo*，内存地址为 *ssss:oooo*。

以上 USER_ROM*n*，USER_ROM*n*_FILENAME，USER_ROM*n*_DEVICEID，USER_ROM*n*_VENDORID，USER_ROM*n*_DESTSEG 以及 USER_ROM*n*_DESTADDR 等字段，都是用来给使用者自定、添加程序代码之用的字段。n 一样从 1～4 有四组位置。

实际 OEM.MAK 范例

本书所建议实作的两块以Sis530 芯片组为结构的主板(丽台 Leadtek WinFast 5310MA，梅捷 Soyo SY-5SSM)，均采用编号 ITE8661F 这块 Super I/O 芯片来处理 COM1，COM2，IR Port(红外线传输端口)，Printer Port (LPT1/EPP/ECP) 等等。我们切换到位于 OEM 目录底下的厂商层(DEMO，DEMO1，DEMO2，...)，打开里面的 OEM.MAK 维护文件来瞧瞧：

```
#*****************************************************************************;
#*****************************************************************************;
# **                                                                         **;
# **          (C)Copyright 1985-1996, American Megatrends, Inc.              **;
# **                                                                         **;
# **                        All Rights Reserved.                             **;
# **                                                                         **;
# **          6145-F Northbelt Pkwy, Norcross, GA 30071                      **;
# **                                                                         **;
# **                        Phone (770)-246-8600                             **;
# **                                                                         **;
#*****************************************************************************;
#*****************************************************************************;
#*****************************************************************************;
# $Header: /BIOS/OEM/GENERIC/INTEL/440LX/oem.mak  3/26/97 7:18p Juand $
#
# $Revision: 2 $
#
# $Date: 3/26/97 7:18p $
# ****************************************************************************;
# ****************************************************************************;
# Revision History
```

以下省略

```
# 1   1/13/97 2:22p Debkumar
# New file for 6.27.00.
#
#**************************************************************************#
# ------------------------------------------------------------------------ #
!INCLUDE io.mak
!INCLUDE ..\..\bios.mak
# ------------------------------------------------------------------------ #
REF_NUMBER     = 009999              # 4 decimal digits for customer        #
#------------------------------------------------------------------------- #
# Source code, ROMID, BIOS-TAG control parameters.                         #
# BIOSTAG (ROMID) will be $(PROJECT_PATH)$(PROJECT_BUILD)                  #
# ------------------------------------------------------------------------#
PROJECT_PATH   = 1SIS5             # 1st digit =  Location                 #
#                  0 = Atlanta, 1 = Taiwan          #
# Last 4 digit is Project Number assigned
# by supervisor.                                    #
PROJECT_BUILD  = 300               # Build Number and should be incremented #
# with any new release.                            #
# Maximum 3 digits.                                #
# ------------------------------------------------------------------------ #
BIOSFILE   = $(PROJECT_PATH)$(PROJECT_BUILD) ←— BIOS 文件名为 1SIS5+300
ALL: PREBUILD $(BIOSFILE).ROM CLEANUP                +.ROM=1SIS5300.ROM
# ------------------------------------------------------------------------ #
#                  MULTIPROCESSOR  SUPPORT                                 #
# ------------------------------------------------------------------------ #
NCPU               = 1 ←———— 指定为单处理器(Single Processor)平台
                             # 1 -> Single Processor BIOS                 #
                             # 2 -> Dual Processor BIOS                   #
                             # n -> Multi Processor BIOS                  #
BUP6               = 0       # 1 = Include P6 support                     #
```

```
# This needs Flash support using 4GB                                    #
# method (FLASH_4GB must be set to 1).                                  #
# MP TABLE SUPPORT - The MP Table should only be built if there is an   #
# I/O APIC on the motherboard.                                          #
MPS_SUPPORT    = 0          # 0=  IO APIC not present in motherboard    #
                            #     (MP table need not to be built)       #
                            # 1 = IO APIC present in motherboard        #
                            #     (MP table need to be built)           #
# If there is an I/O APIC on the motherboard, then the MP Table may be  #
# modified to include entries for PCI interrupts in INTIN#'s above the  #
# standard IRQ0 - IRQ15. The routing of PCI interrupts is somewhat      #
# hardware specific so see the MPPOST.ASM file for more detail.         #
# EXTENDED_APIC switch is valid only if MPS_SUPPORT switch is set to 1. #
EXTENDED_APIC = 0           # 0 = hardware not using INTIN# lines of    #
                            #     IO APIC above 15 for PCI slots.       #
                            # 1 = hardware using INTIN#15-INTIN#22      #
                            #     as PCI slot interrupt routing and     #
                            #     INTIN#23 as SMI.                      #
OEM_ID         = SiS        <-------- 指定 OEM ID 为 SiS

PRODUCT_ID     = SiS530     <-------- 指定 Product ID 为 SiS530

                            以下省略

BIOS_SIZE      = 2          <-- 指定这块 SiS530 主板 BIOS 容量为 256KB
SOFT_OFF       = 1          <-- 启用 ATX 软件电源开关
ACPI_SUPPORT   = 1          <-- 启用 ACPI 电源组态接口管理
ONCHIP_LM78    = 1
LM78_BASE_ADDRESS = 295h
LM75_SUPPORT   = 1
LM75_SLAVE_ADDRESS = 9eh
SILENT_BOOT    = 0          # 1 = Silent Boot Support                  #
GPNV_SUPPORT   = 1          # 1 = GPNV support                         #
GPNV_FLASH     = 1          # 0 = GPNV in IO Mapped NVRAM              #
```

```
EVENT_LOG               =0
USB_SUPPORT             =1          # 1 -> USB Support present                #

# SUI_FILE              =hf130.bin
# SUI_FILE              =easy132.BIN
# SUI_FILE              =HF122.BIN
# SUI_FILE              =Win29.BIN

SUI_FILE                =SP130.BIN ◄— 指定 SP130.BIN 为 Setup 接口程序
# SIGN_ON_MSG1          ='SIS-530 BIOS Version 1.0 ', 0   ◄— BIOS 在 POST 阶段
                                                          所显示的信息字符串
# ------------------------------------------------------------------------------ #
INTEL       =0                                                                   #
SST         =1                                                                   #
ATMEL       =1                                                                   #
WINBOND     =1                                                                   #
AMD         =1                                                                   #
# ------------------------------------------------------------------------------#
# SMBus Base Address to be programmed in the PIIX4 Function 3, Reg 90..91
# This Base Address will be programed in Boot Block code
# !!!! DO NOT CHANGE THESE VALUES !!!!
# ------------------------------------------------------------------------------ #
SMB_BASE_HIGH   =004h
SMB_BASE_LOW    =040h
# ------------------------------------------------------------------------------#
# PM (ACPI) Base Address to be programmed in the PIIX4 Function 3, Reg 40..41
# This Base Address will be programed in Boot Block code
# !!!! DO NOT CHANGE THESE VALUES !!!!
# ------------------------------------------------------------------------------#
PM_BASE_HIGH    =004h
PM_BASE_LOW     =000h
# ------------------------------------------------------------------------------#
# Include the OBJ.MAK file defining the OBJ Groups based on make file            #
# switches defined in BIOS.MAK and in this file above.                           #
```

```
# ========NOTE====NOTE====NOTE========                                    #
# NOTE : All the OEM specific Make file switches MUST BE DEFINED above.   #
# ------------------------------------------------------------------------#
!INCLUDE ..\..\obj.mak
# ------------------------------------------------------------------------#
$(BUILD_DIR)\oemmake.equ: $(MKF)
copy << $(BUILD_DIR)\oemmake.equ
; Generate new makefile equates using OEM specific switches
```

以下省略

```
# ------------------------------------------------------------------------ #
# Silent Boot Support                                                      #
# If Silent Boot is supported, OEM Logo Module should be present,          #
# specify the OEM Logo module binary imgae filename including the          #
# pathname here.                                                           #
# ------------------------------------------------------------------------#
!if $(SILENT_BOOT)
OEM_LOGO_FILENAME    =              ← 目前此字段为空白
!endif
# ------------------------------------------------------------------------#
# Multi Language Support                                                   #
# ------------------------------------------------------------------------#
ML_MODULE                 = 0    # 1 = Multi-Language Module present      #
ML_MODULE_FILENAME    =      # Multi Language Module Filename.Ext          #
ML_MODULE_DESTSEG     =      # Multi Language Module Dest Segment          #
ML_MODULE_DESTADDR = # Multi Language Module Dest Addr                    #
# If Multi Language Module is present,   #
#    then      1. Generate the Assembly file in Project directory.        #
#              2. Write the Dependencies here.                            #
#              3. Write the command to assemble the file and create the   #
#           OBJ files in BUILD_DIR. #
#              4. Create the Module Binary File in BUILD_DIR.             #
```

加载 SiS530 显示芯片的 BIOS 程序代码

```
# ------------------------------------------------------------------------------ #
# Specify the user defined ADDON ROMs to be put into the final ROM file.#
# These ADDON ROMs binary files must be present in ADDON_ROM directory.      #
# ------------------------------------------------------------------------------ #
PCI_ROM1              = 1          # 1 = PCI ROM1 present in ADDON_ROM dir    #
PCI_ROM1_FILENAME = 530v107.rom  # PCI ADDON ROM1 Filename.Ext
PCI_ROM1_DEVICEID     = 6306     # PCI ADDON ROM1 Device ID                  #
PCI_ROM1_VENDORID = 1039         # PCI ADDON ROM1 Vendor ID                  #
PCI_ROM2              = 0        # 1= PCI ROM2 present in ADDON_ROM dir      #
PCI_ROM2_FILENAME =              # PCI ADDON ROM2 Filename.Ext               #
PCI_ROM2_DEVICEID =              # PCI ADDON ROM2 Device ID                  #
PCI_ROM2_VENDORID =              # PCI ADDON ROM2 Vendor ID                  #
PCI_ROM3              = 0        # 1 = PCI ROM3 present in ADDON_ROM dir     #
PCI_ROM3_FILENAME =              # PCI ADDON ROM3 Filename.Ext               #
PCI_ROM3_DEVICEID =              # PCI ADDON ROM3 Device ID                  #
PCI_ROM3_VENDORID =              # PCI ADDON ROM3 Vendor ID                  #
PCI_ROM4              = 0        # 1 = PCI ROM4 present in ADDON_ROM dir     #
PCI_ROM4_FILENAME =              # PCI ADDON ROM4 Filename.Ext               #
PCI_ROM4_DEVICEID =              # PCI ADDON ROM4 Device ID                  #
PCI_ROM4_VENDORID =              # PCI ADDON ROM4 Vendor ID                  #
USER_ROM1             = 0        # 1 = USER ROM1 present                     #
USER_ROM1_MODULEID =             # User ROM1 Module ID (e.g. 29, 2A, etc.)   #
USER_ROM1_FILENAME =             # User ROM1 Filename.Ext                    #
USER_ROM1_DESTSEG =              # User ROM1 Destination Segment             #
USER_ROM1_DESTADDR =             # User ROM1 Destination Address             #
USER_ROM2             = 0        # 1 = USER ROM2 present                     #
USER_ROM2_MODULEID =             # User ROM2 Module ID (e.g. 29, 2A, etc.)   #
USER_ROM2_FILENAME =             # User ROM2 Filename.Ext                    #
USER_ROM2_DESTSEG =              # User ROM2 Destination Segment             #
USER_ROM2_DESTADDR =             # User ROM2 Destination Address             #
# ------------------------------------------------------------------------------ #
!INCLUDE ..\..\core.mak          <-- 引入核心层的 core.mak 文件
# ------------------------------------------------------------------------------#
```

1SIS5300.ROM 是编译出来的ROM BIOS图像文件名，而 MAKE.OUT 记录着编译过程中的一切信息，当你发现编译过程可能有错误时，可以用任何一个文本编辑程序，打开 MAKE.OUT 这个记录文件来查找哪个环节出了问题。

CHAPTER

5

BIOS的内部模块与执行流程介绍

在前面的章节，我们深入地介绍了 AMIBIOS 的目录规划，以及重要编译 / 维护所要改动的文件。现在让我们深入 BIOS 的内部核心，讨论它们的模块，以及开机时的执行流程；最后浏览几个关键性的 BIOS 原始程序代码，相信能让读者更容易了解、掌握 BIOS 的执行过程。

5-1 BIOS 内部模块的说明

BIOS 影像文件 (Image File) 的长相，或者说存储在 FlashROM 芯片的真实形式，究竟是如何呈现出来的呢？

如果切换到 OEM 厂商层，然后输入 AMIBCP 1sis5300.rom 的 DOS 命令，选择 Edit BIOS module(编辑 BIOS 模块)，可以看到如图 5-1 所示的画面。

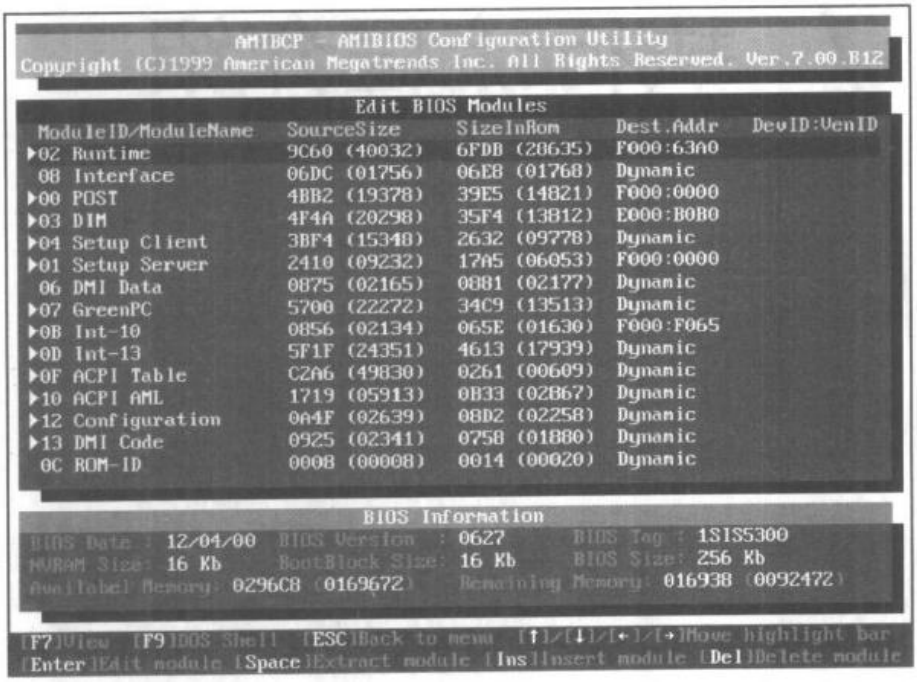


图 5-1 以 AMIBCP 所查寻的 BIOS 各内部模块

如果读者是打开 make.out (编译 BIOS 过程记录文件)，会在最后一个 Page 看到更多的模块信息，如图 5-2 所示：

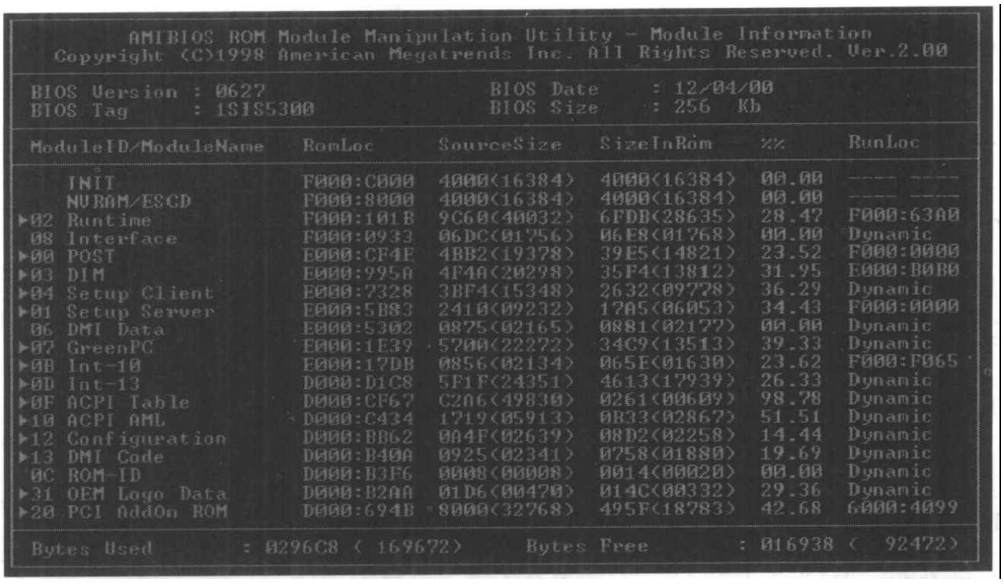


图 5-2 从 make.out 所查出的各模块名称、ID、是否压缩与解压地址等信息

我们将 AMI BIOS 内部的模块相关资料，包含它的 ID 号码可否修改，以及在运行时是否先解压缩的原则列举如下：

AMIBIOS 内部模块 ID、可改动特性与模块用途

模块名称	ID 号码	可否修改	是否压缩	模块用途
INIT		否	否	从 Reset 跳入点开始的 Boot Block 程序代码
NVRAM/ESCD		否	否	FlashEPROM 的 GPNV 区域, 存储系统组态数据
POST	00h	否	否	POST 程序代码
Setup Server	01h	否	是	AMIBIOS 设置服务器端
Runtime	02h	否	是	BIOS 功能函数 / 常驻的程序代码
DIM	03h	否	是	Device Initialize Manager 装置初始化

Setup Client	04h	可	是	提供 AMIBIOS 设置画面的接口程序
Remote Server	05h	可	是	搭配远程遥控除错系统(Remote Debugger)
DMI Data	06h	可	是	Desktop Management Inteface 数据区
GreenPC	07h	否	是	符合 APM 规范的节电管理程序
Interface	08h	否	否	BIOS 模块压缩 / 解压子程序模块
MPS	09h	可	是	多重处理器(Multi-Processor, MP)起始模块
Int-10	0Bh	可	是	BIOS 文字 / 绘图等显示处理模块
ROM-ID	0Ch	可	否	记录 ROM ID 与相关信息的模块
Int-13	0Dh	可	是	BIOS 软盘 / 硬盘 / 光驱读写形式
ACPI Table	0Fh	可	是	ACPI 规范之控制表格区
ACPI AML	10h	可	是	ACPI 之 AML 句柄存放区域
P6 Micro Code	11h	否	是	Intel P6 处理器微码修补模块
Configuration	12h	可	是	系统组态显示模块
DMI Code	13h	可	是	DMI 控制程序码模块
PCI AddOn ROM	20h	可	是	PCI 适配卡 ROM 程序代码模块
Multilanguage	21h	可	是	多重语言控制模块
Font DataBase	30h	可	是	中文(双位语系)字形数据模块
OEM Logo Data	31h	可	是	屏幕右上方 OEM 商标画面模块
Graphic Logo Code	32h	可	是	Silent Boot, OEM 全屏幕画面处理程序
Graphic Logo Data	33h	可	是	Silent Boot, OEM 全屏幕图形文件数据模块
Virus	36h	可	是	趋势 Trend Chip Away Virus(TCAV) 模块

这每一个模块，事实上是由好几个甚至几十个编译过的 *.ASM，*.OBJ 等文件所连结而成，至于每个模块会参考到哪些 *.ASM、*.OBJ 或其他文件，这则是由 AMI 当初设计的一连串编译维护批处理文件，如 MAKEBIOS.BAT，MAKEALL.BAT，以及位于芯片层的几个 *.MAK 维护文件，依照不同的芯片组特性，以及前面介绍过的 IO.MAK、OEM.MAK 等文件中字段 / 标志位的控制来决定，模块内会牵扯到的 *.ASM，*.OBJ 等文件内容是随时有所变动的 。

由于本书篇幅有限，笔者并不对哪个模块是由哪些 *.ASM、*.OBJ 所组成的来龙去脉详细说明；有兴趣的读者，可以自己从光盘中，对这些阵仗庞大的各式 *.BAT、*.MAK 去自行 trace ，相信假以时日，对读者编写 *.mak 文件的功力必然有所提高。

BIOS 程序代码能屈能伸 / 能压缩就压缩

BIOS 模块阵仗中，INIT、NVRAM/ESCD、POST 以及 Interfacce 这四个模块是属于开机最初被执行的模块，也是所有 BIOS 模块中，唯一以非压缩的程序代码形式(Un-compressed code)存储于 ROM，也就是说 BootBlock 的程序代码跟存储在FlashROM 的程序代码是完全相同的；而其他模块在编译 / 连结完成之后，要做模块总连接成一块 BIOS ROM 之前，事实上都是先经过压缩处理，然后才写回 BIOS ROM 文件内。

为什么要压缩呢？说穿了，是为了降低主板采用的 FlashROM 芯片的容量(成本)。因为 Flash 内存是每块主板不可缺少的重要组件，在成本支出中占有一定比例，而 Flash 内存容量越大，价格就越贵。

计算机主板所采用的 FlashROM 芯片，容量是以 128KB x 2^n (n=0, 1, 2, ...) 开始计算，从以前都是以最早的 128KB (1Mb)、256KB (2Mb)，到目前最高容量的 512KB (4Mb)，容量每加一倍价格就增加许多，主板厂家的心态就是“能用越小容量的 FlashROM，塞入应有的功能越好”。

如果某家主板厂商开发一款主板，原始设计有 10 项规格、20 项功能，当 compile 出来的程序代码长度是 160、170K，再加上 ESCD、PnP 等固定区域，总和超过 256KB，如果 BIOS 内部模块不压缩，就得用 512KB (4Mb) 的 Flash 内存，成本必然会增加许多。

如果使用压缩技术，有需要被调用、执行的时候，再解压缩并 Copy 到 Shadow RAM 或其他特定的内存地址，那 FlashROM 只要放一些没办法压缩的紧急程序(如 BootBlock)、解压缩程序；其他还没调用、使用到的部分可以压缩起来，等到需要时才被加载进来解压缩、还原到原先程序文件。上述的 160、170K 程序代码，压缩一下也许只要 80、90K 左右，连同 ESCD，PnP，DMI 区域，以及附加的适配卡程序代码，也许可以塞进 256KB (2Mb) 的内存。

假定一块 256KB FlashEPROM 芯片比 512KB 芯片便宜两美元，某三大主板厂商月产量能高达一百万片，光是决定采用 256KB 或 512KB 内存芯片，每个月的成本差距就高达两百万美元，由此可以想见 BIOS 惜 byte 如金的情形，对于厂商产品的成本支出会有多大的影响！

回归模块的讨论主题。每一个模块长度，受限于在真实模式下的寻址能力，一般建议单一模块长度控制在 64KB 以下，无论这个模块的压缩率如何，还原回原始未压缩的模块总长度状态，也不可以超过 64KB，否则 BIOS 程序代码开始执行解压程序时，可能会有无法预料的结果，这点是读者在添加新型模块，甚至自行开发程序代码模块以加入 BIOS 时必须要注意的。

5-2 BIOS 执行流程图

BIOS 的发展特性，其实就是在“兼容性”及“近代规格不断累加”的双重因素下，历经十几年的演变而得出现在的风貌；为了扩充、选择上的弹性设计，实在不太可能只用一组或者少数几组*.ASM 文件，下几个简单的指令，就可以编译、连结出来一套功能完整的 BIOS，能吃下这么多的历史规格，以及幕后的服务功能。

光从本书所附的源代码光盘中，里面所涉及到的源代码(*.ASM)、目标文件(*.OBJ)也超过一千多个，程序代码的执行过程更是跳来跳去，要一一地查看、浏览，任何再厉害的汇编语言高手，也不免有陷入茫茫程序代码海，或浩瀚程序代码沙漠中的无边无尽之感。

在本书第 3 章，第三节 BIOS 诊断码 (Post Code) 简介中，我们知道当初设计 POST CODE 的用意，就是为了加快系统的开发与除错速度，缩小问题发生的范围。如果我们以这个0～FF 约一百多个诊断码为执行过程的主要节点(Node)，为这个茫茫 BIOS 程序代码海域划分成一百多个坐标点，那么在浏览、观察源代码的过程中，就会比较快速而有所掌握。而这也是日后维护、除错时所必须要掌握的快速索引方式。

在 AMI 公司的同意下，本书第一次独家揭露 Post Code 的节点方式的整个 BIOS 的完整执行流程图。整个图文大体是以英文为主，并注明到参考到的源代码，经过笔者的增修并加注一些中文的批注而成；即使读者是维护其他芯片组版本的 AMI BIOS，也应该可以从这份执行流程图表中获得必要的帮助。

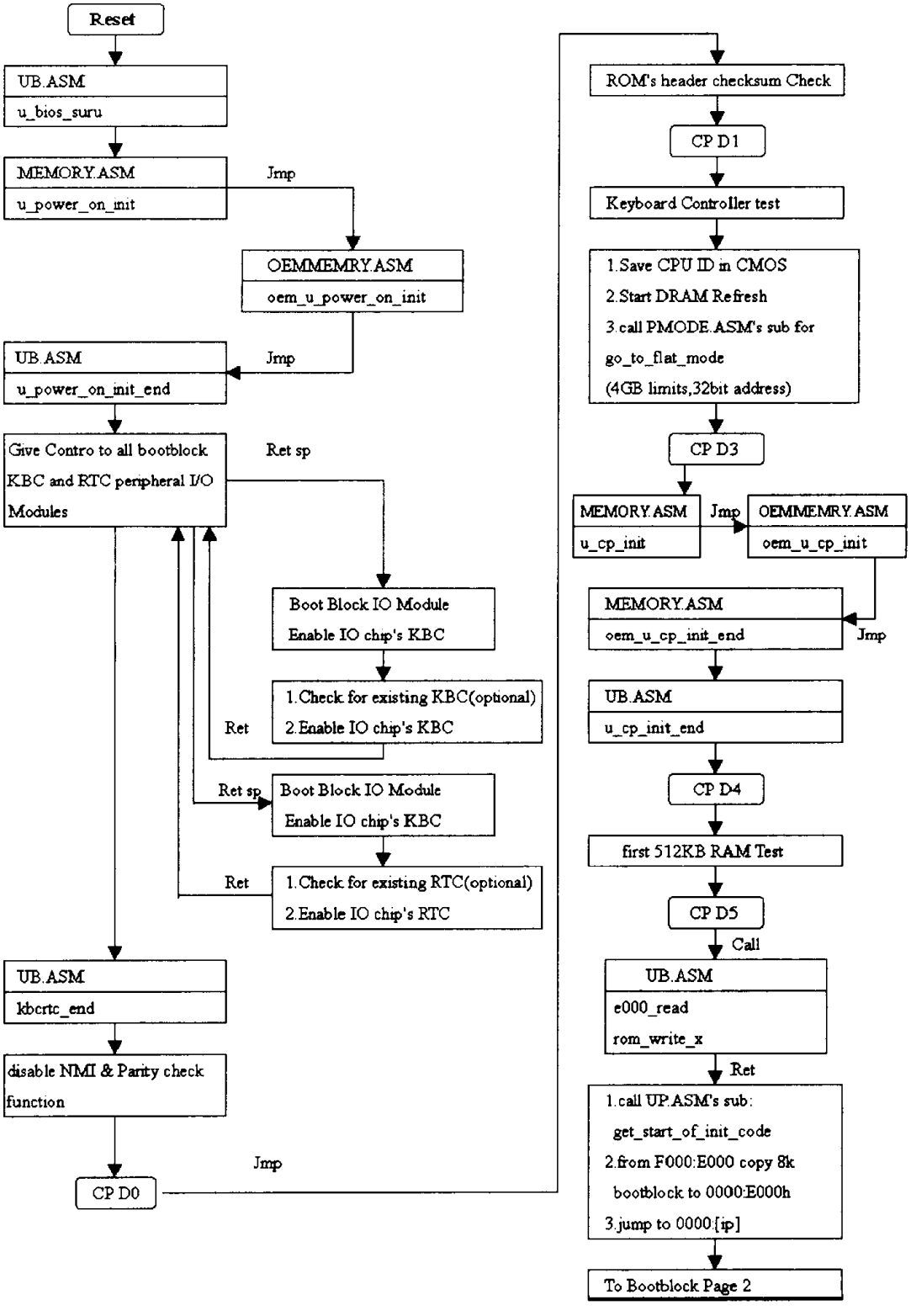


图5-3 Bootblock Page 1(Page 1 of 14)

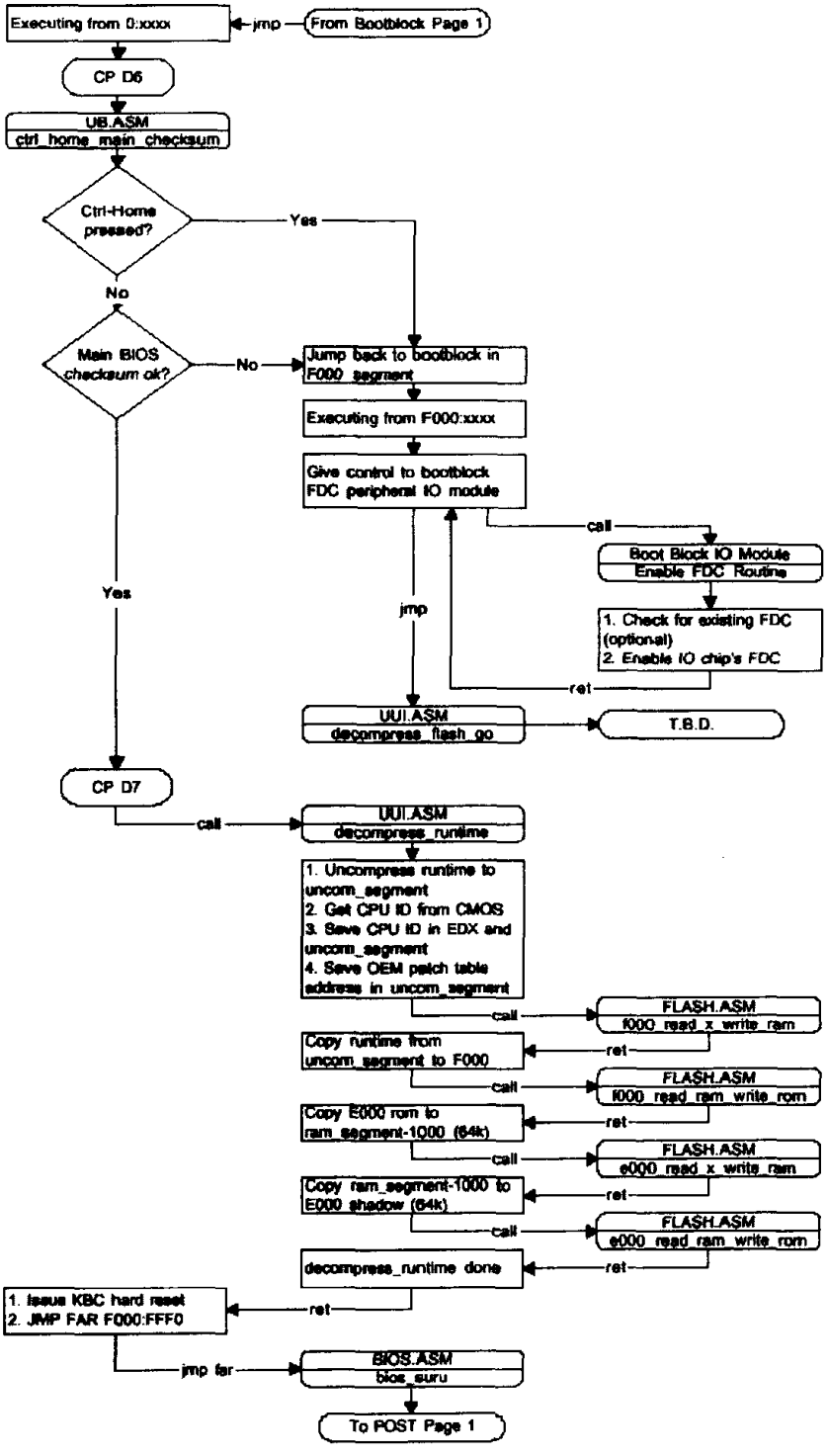


图5-4 Bootblock Page 2(Page 2 of 14)

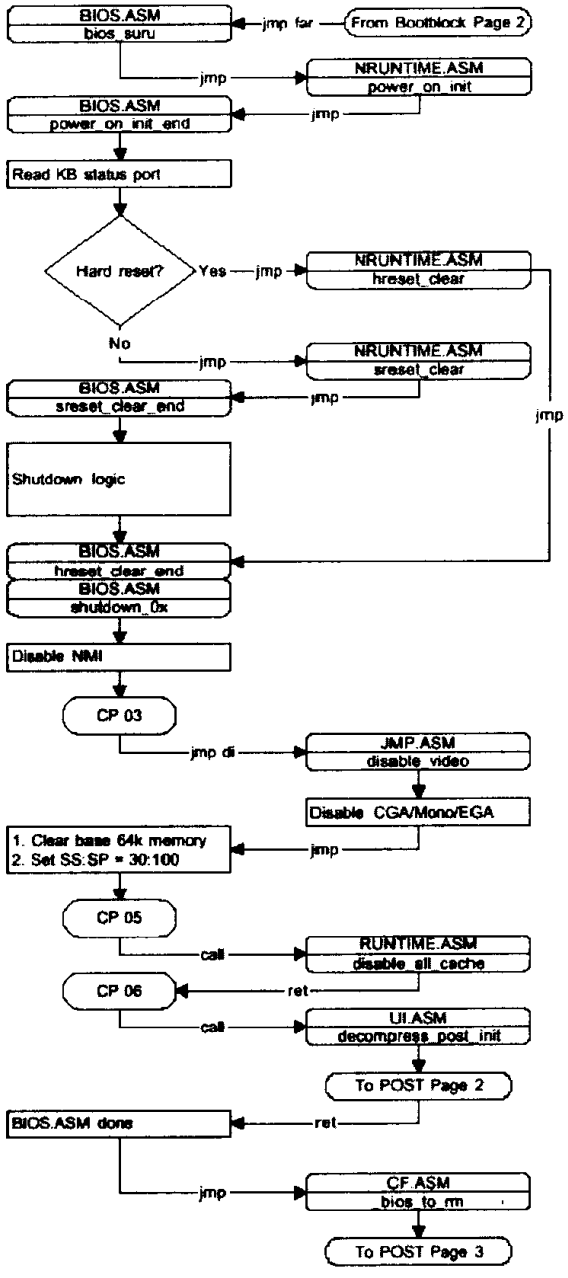


图5-5　POST Page 1(Page 3 of 14)

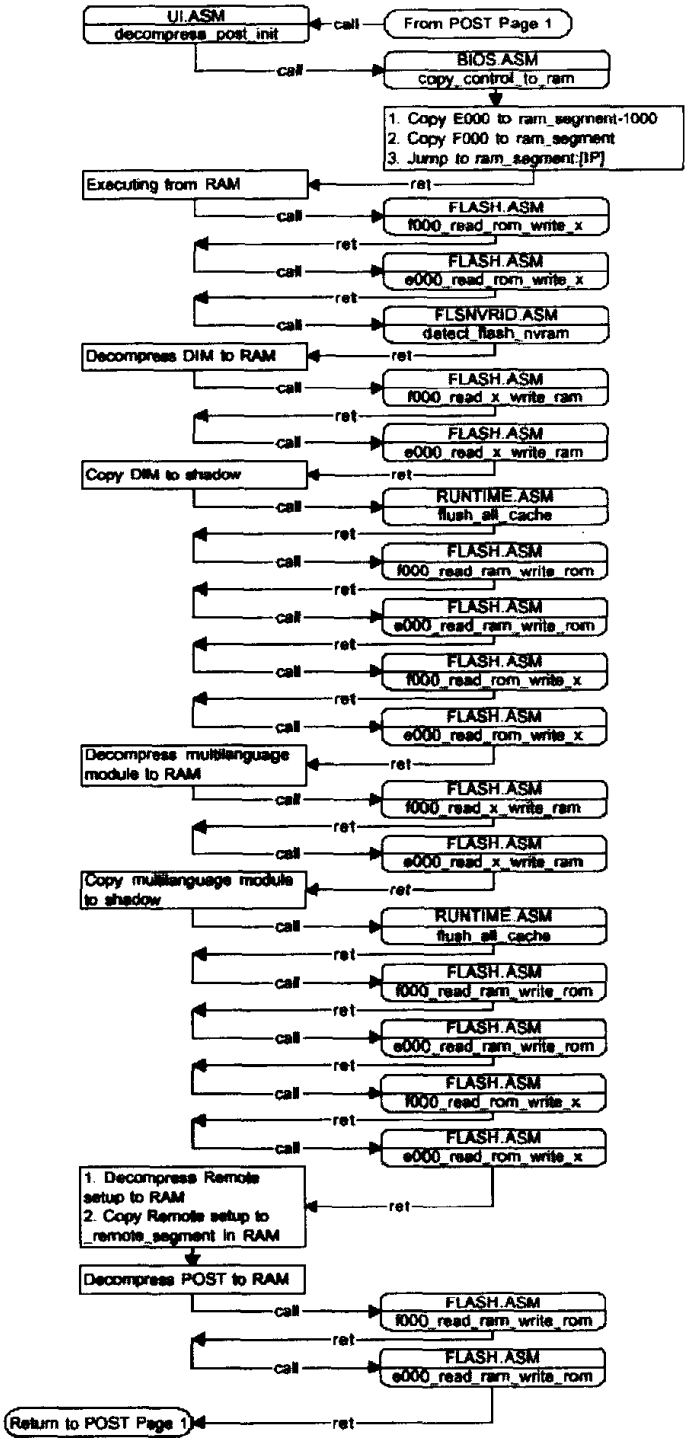


图5-6 POST Page 2(Page 4 of 14)

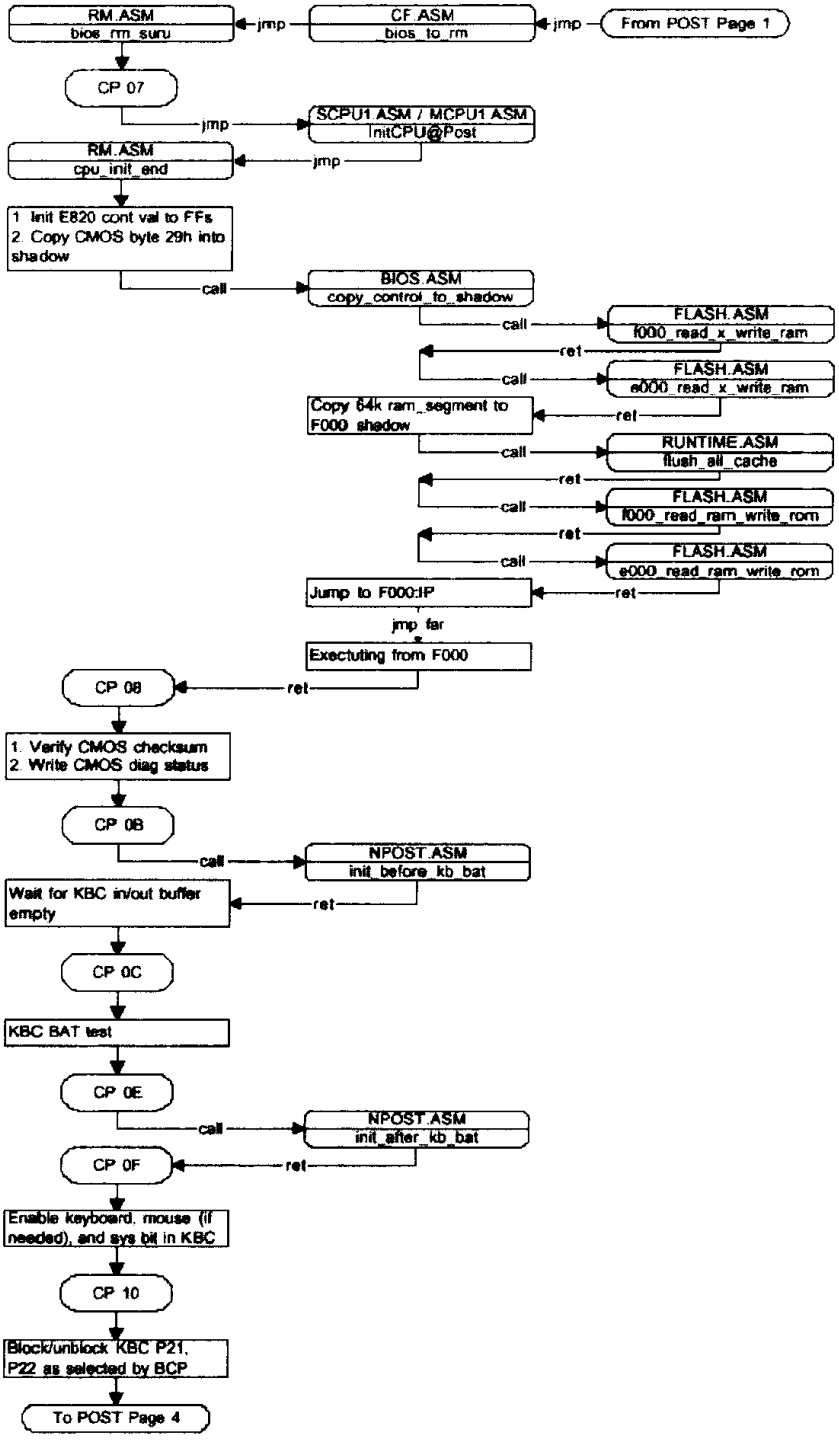


图5-7　POST Page 3(Page 5 of 14)

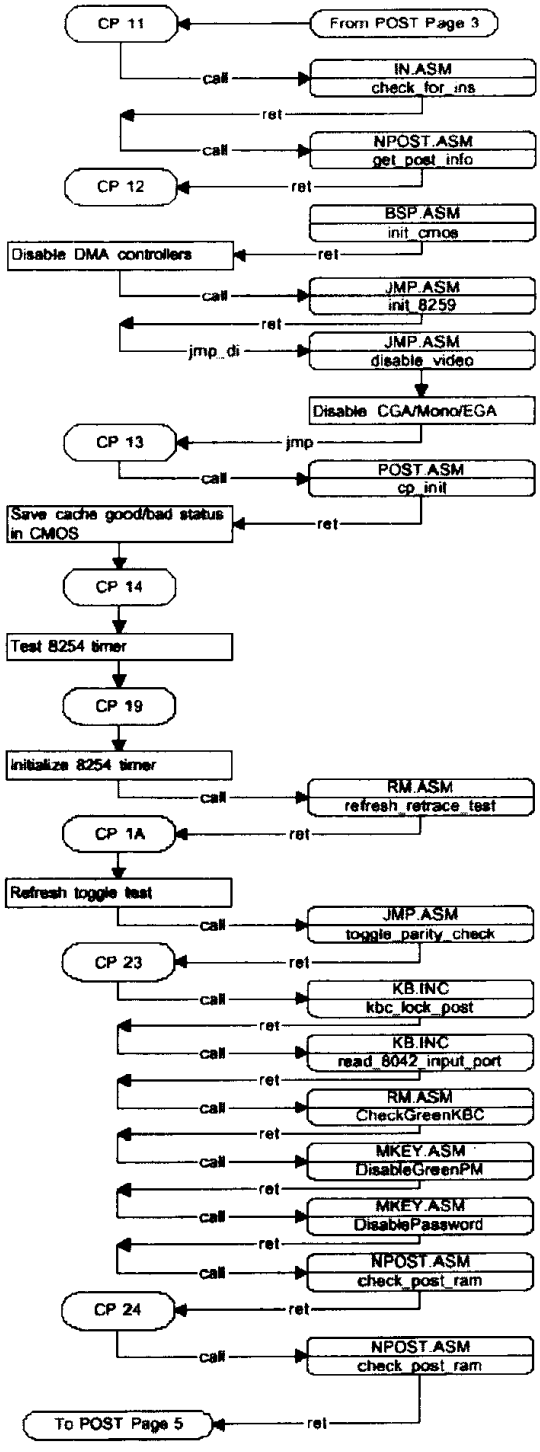


图5-8 POST Page 4(Page 6 of 14)

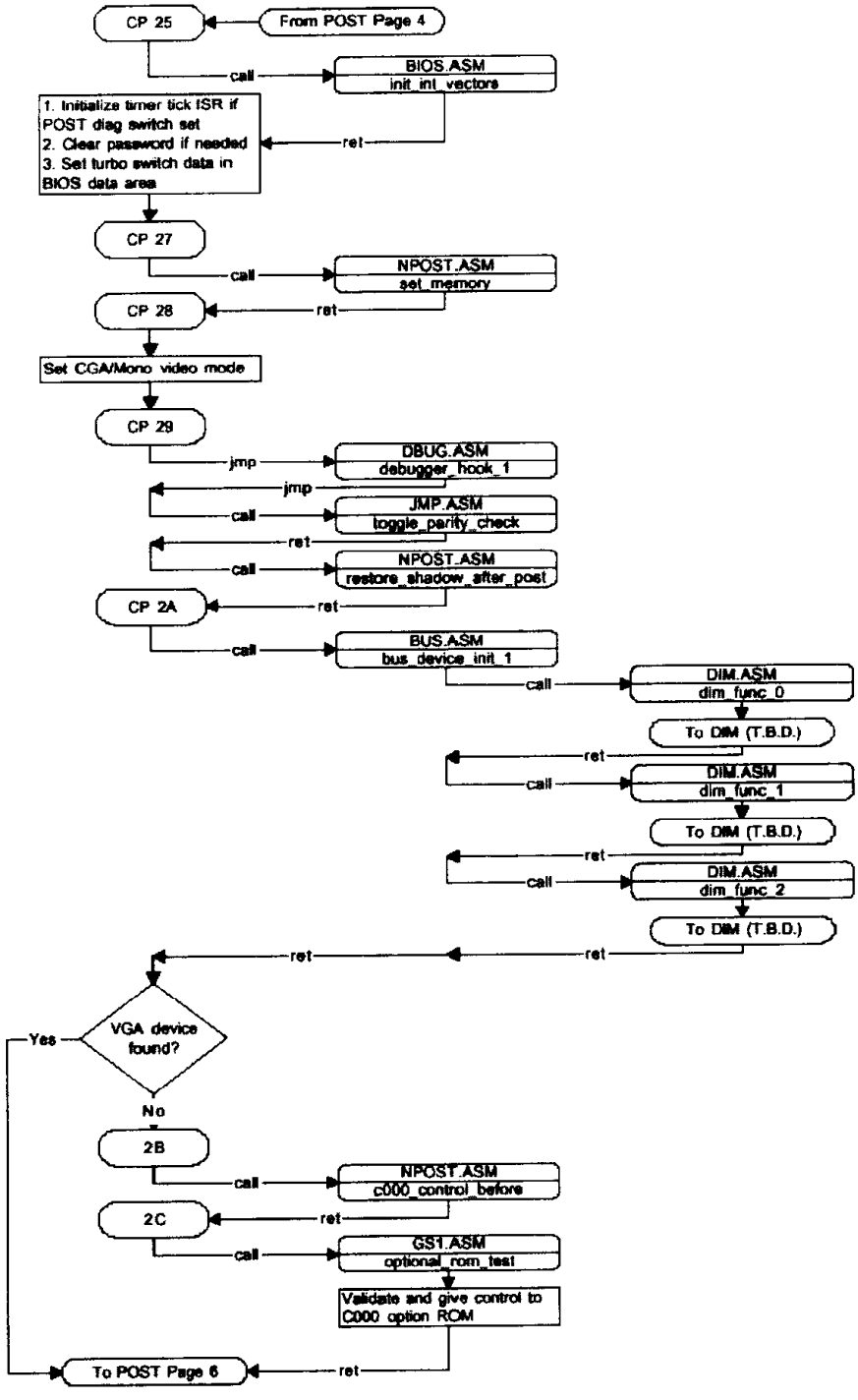


图5-9　POST Page 5(Page 7 of 14)

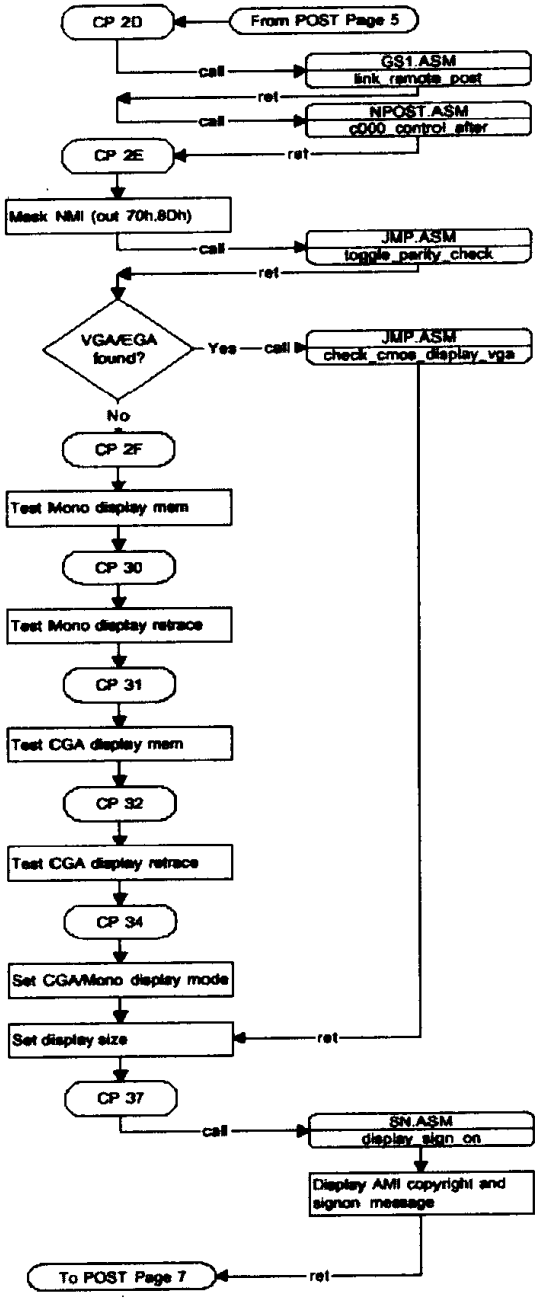


图5-10 POST Page 6(Page 8 of 14)

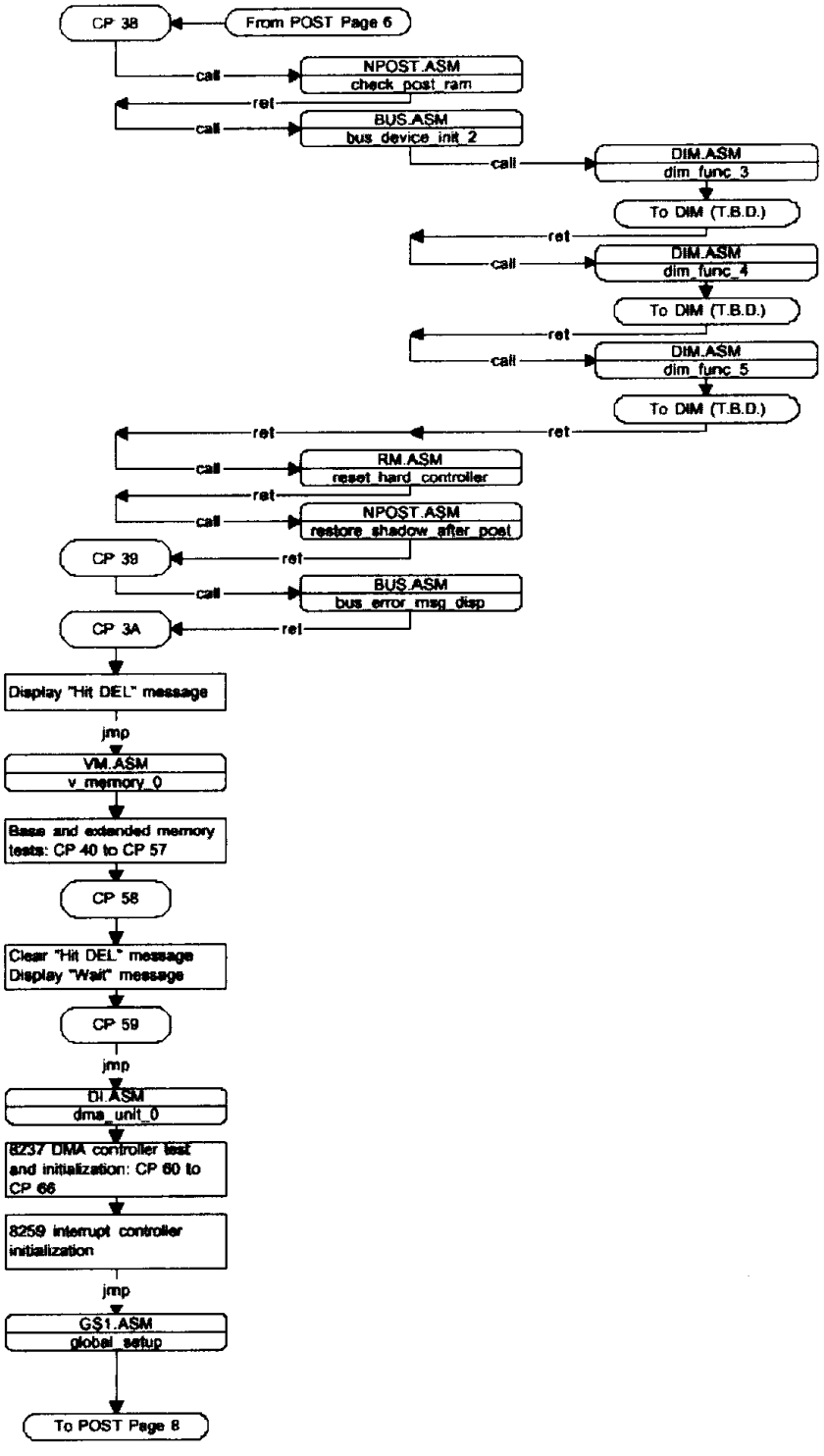


图5-11　POST Page 7(Page 9 of 14)

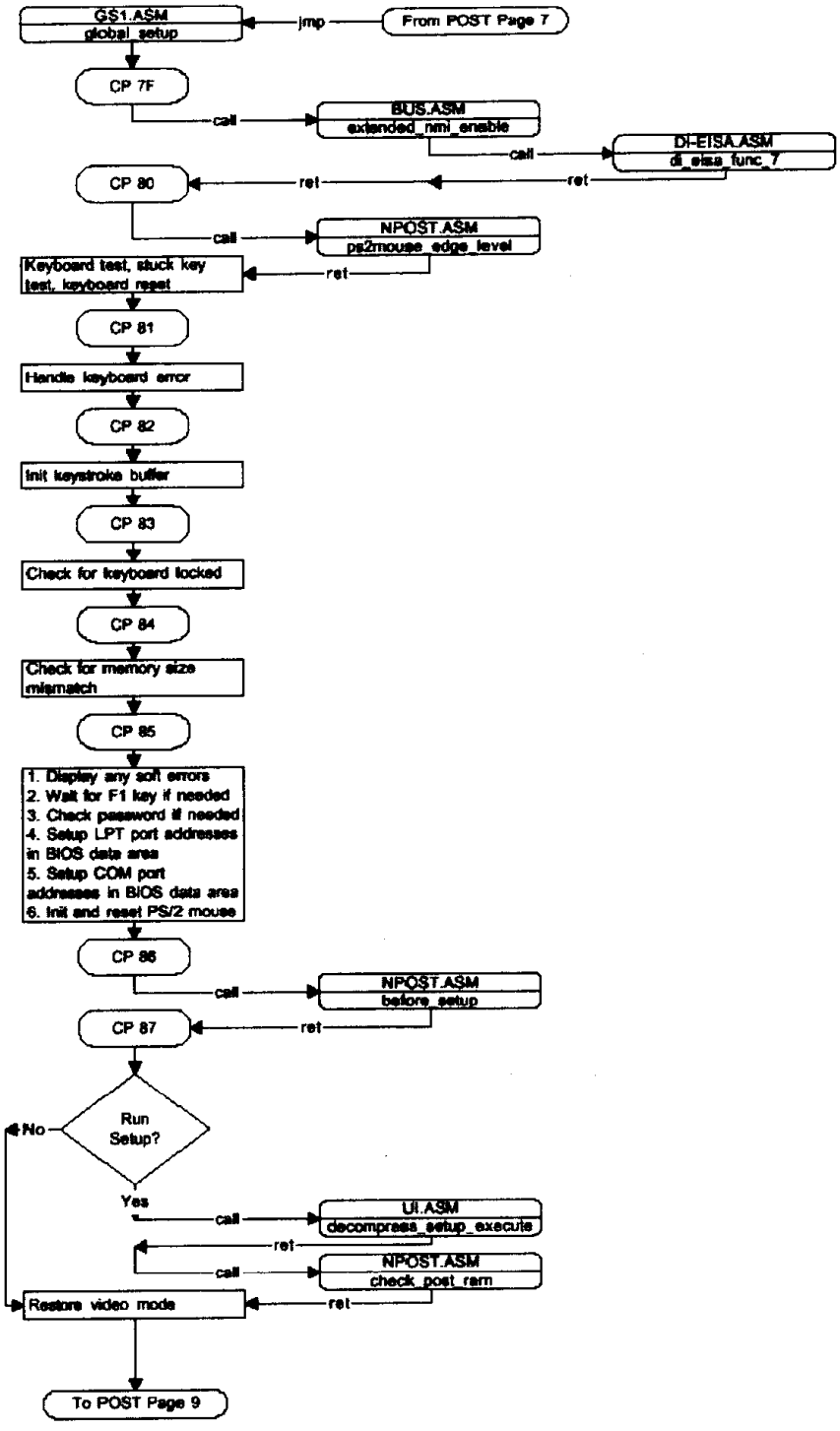


图5-12 POST Page 8(Page 10 of 14)

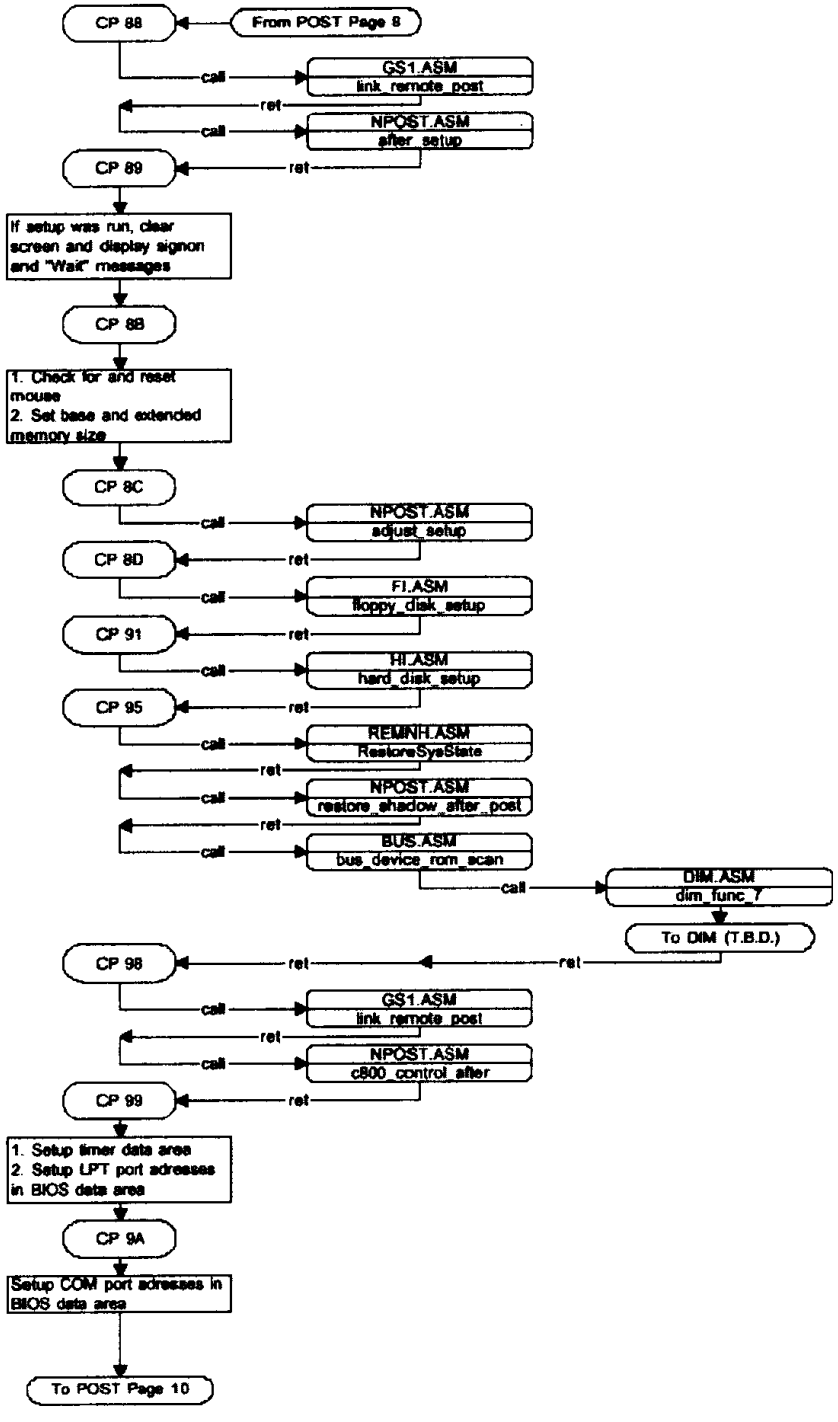


图5-13 POST Page 9(Page 11 of 14)

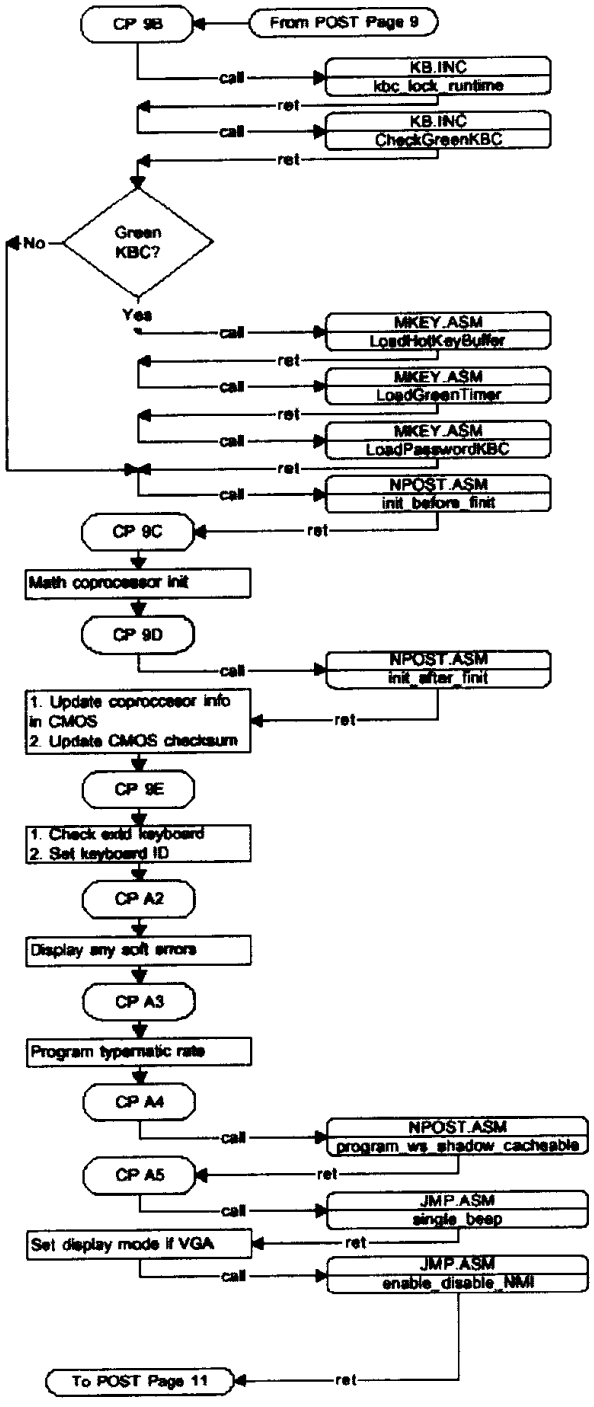


图5-14 POST Page 10(Page 12 of 14)

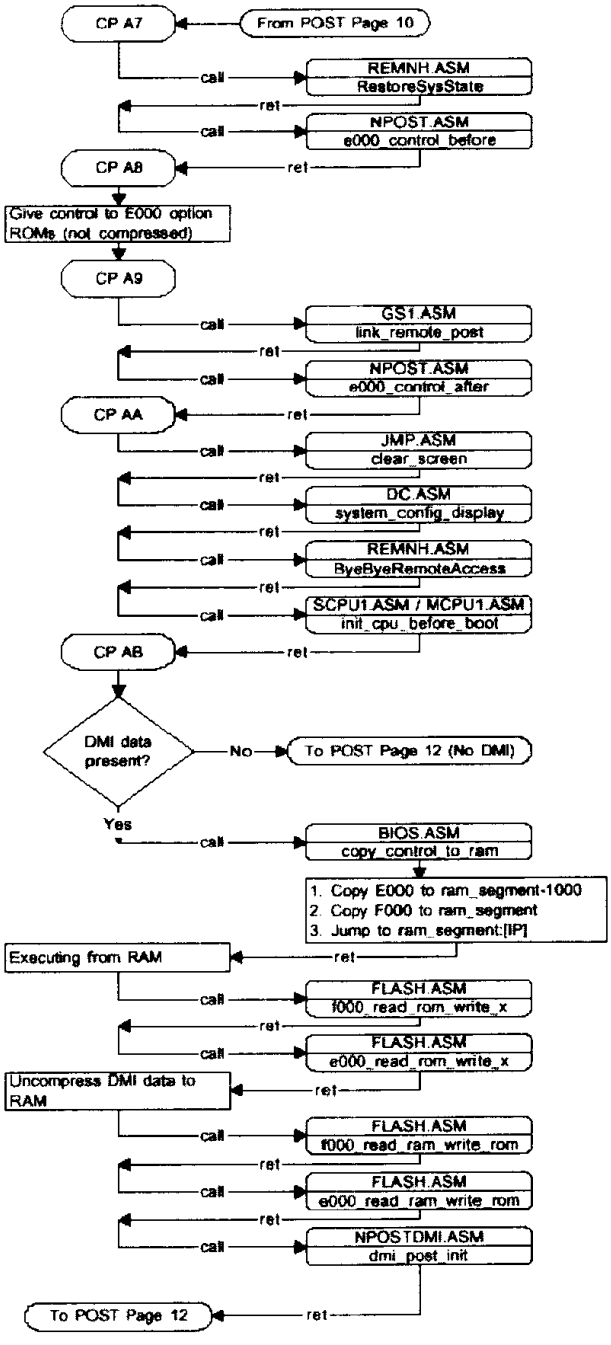


图5-15　POST Page 11(Page 13 of 14)

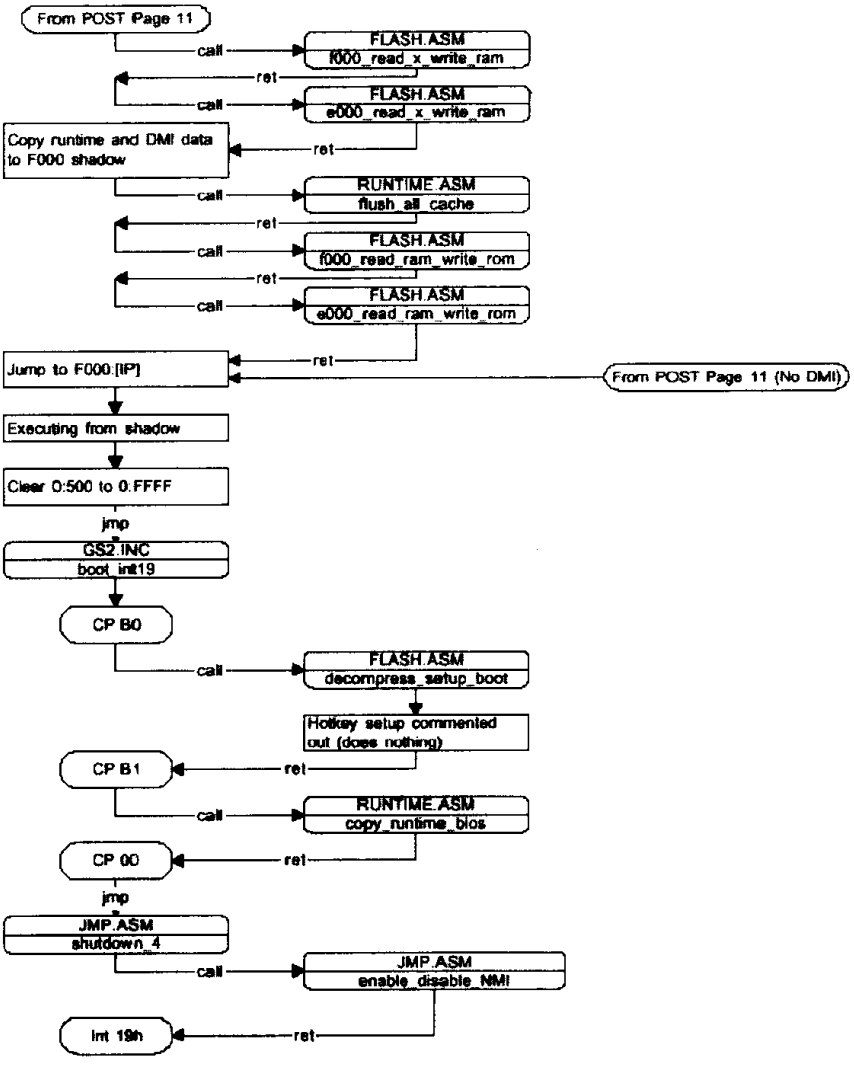


图5-16　POST Page 12(Page 14 of 14)

5-3 BIOS 程序源代码浏览(一) —— BootBlock(激活区块) 的相关模块

现在，就让我们从第三章介绍过的 POST Code 显示号码的顺序，来真正地浏览几个关键性的 BIOS 源代码 (*.ASM) 吧！

文件名	UB.ASM
路径	\627.10\BOOTINI\UB.ASM 或..\..\..\BOOTINI\UB.ASM (相对于OEM 厂商层)

```
page  60,132
title  UNCOMPRESSED BIOS START CODE(UB.ASM)
```

以下省略

```
;--------------------------------------------------------------------------------;
        include mbiosequ.equ    ;  加载BIOS相关定义常数
        include mbiosmac.mac    ;  加载bios编写时用到的宏指令定义
        include bblkinit.equ    ;  关于bios区块定义的引入文件
;--------------------------------------------------------------------------------;
        public u_bios_suru                  ;  将本程序代码 _bios_suru、
        public ptr_to_source_buffer         ;  ptr_to_source_buffer 等标记
                                            ;  均声明为Public
        extrn  u_power_on_init:near         ;
        public u_power_on_init_end          ;

        extrn  u_cp_init:near               ;
        public u_cp_init_end                ;

        extrn  rom_filename:byte            ;
        extrn  boot_block_flag:byte         ;
        extrn  refresh_low_value:byte       ;
        extrn  bootblock_fdc_table:byte     ;
```

```
        extrn bootblock_fdc_table_end:byte      ;
        extrn bootblock_kbc_rtc_table:byte      ;
        extrn bootblock_kbc_rtc_table_end:byte  ;
        extrn go_to_flat_mode:near              ;
        extrn fixed_delay:near                  ;
        extrn search_compressed_module:near     ;
        extrn ib_free:near                      ;
        extrn cmos_data_in:near                 ;
        extrn cmos_data_out:near                ;
        extrn restore_cpuid:near                ;
        extrn check_ctrl_home:near              ;
        extrn check_main_bios_checksum:near     ;
        extrn form_rom_image:near               ;
        extrn get_start_of_init_code:near       ;
;-------------------------------------------------;
        extrn int_0e:word                       ;
        extrn diskette_parameter_table:word     ;
        extrn int_40:word                       ;
        extrn init_8237_dma:near                ;
        extrn enable_internal_cache:near        ;
        extrn disable_internal_cache:near       ;
        extrn  flash_program:near               ;
        extrn display_blink_halt:near           ;
        extrn display_on_port:near              ;

        extrn floppy_cntlr_err:byte             ;
        extrn file_abs_msg:byte                 ;
        extrn flp_ins_msg:byte                  ;
        extrn read_err_msg:byte                 ;
        extrn flp_read_msg:byte                 ;
;-------------------------------------------------;
cgroup  group _text
_text   segment word  public 'CODE'
        assume  cs:cgroup
.486p
;-------------------------------------------------;
```

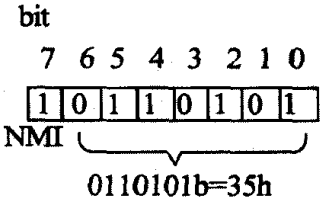
均设为外部调用的子程序，需要链接其他已声明公开的目标代码(.obj)

均设为外部调用的子程序，需要链接其他已声明公开的目标代码(.obj)

```
        public _UB_STARTS
_UB_STARTS   label byte      ; 标示本程序代码(UB.ASM)的开始标记
;--------------------------------- ; 刚开机从FFFF:0000跳到F000:E05Bh程序地址
u_bios_suru:                 ; 后就会跳到u_bios_suru处(约F000:0C00 X)
        cli                  ; 马上将中断允许标志位关掉,禁止硬件中断产生
        cld
        mov   ax,cs
        mov   ss,ax          ; 暂时设置SS=CS=0F000h
        jmp   u_power_on_init ; 跳至F000h:offset u_power_on_init 处的地址
u_power_on_init_end:
;---------------------------------------  ; 检查是否主板芯片组已经内建RTC,
;  ENABLE RTC, KBC if controlled by       ; KBC,若是则进行下一步;若否则先跳到
;       on-board I/O chipset..            ; 专属的 SuperI/O 的子程序,先做 initialize 再
;---------------------------------------  ; 进行下一步
        mov     si,offset cgroup:bootblock_kbc_rtc_table
kbcrtc_00:
        cmp     si,offset cgroup:bootblock_kbc_rtc_table_end
        jae     kbcrtc_end
        lodscs:word ptr [si]             ; AX = routine offset
        mov     sp,offset cgroup:kbcrtc_01
        jmp     ax                       ; do not destroy EDX, SI, SS, SP
        even
kbcrtc_01:
        dw      offset cgroup:kbcrtc_02
kbcrtc_02:
        jmp short kbcrtc_00
kbcrtc_end:
;---------------------------------------  ;
;    Disable NMI and PARITY              ;
;---------------------------------------  ;
        mov     al,0ch
        out     61h,al                   ; disable parity

check_point_ini 0d0h                     ; 送出POST 检查码 0D0h到Port 80h
mov   ebp,edx                            ; store bit31-16 of EDX in bit31-16 of EBP
;---------------------------------------  ;
```

```
;   VERIFY CHECKSUM                    ;   检查Boot Block程序加总检查码
;-----------------------------------------;   (checksum)
        ret_sp get_start_of_init_code   ; DS:DI = ptr to bios header+1
                                        ; DS:DI = ptr to start of init code
        mov     si,di                   ; start offset of init code
        mov     cx,di                   ; start offset of init code
        neg     cx                      ; CX = init code size in bytes
        shr     cx,3                    ; CX = init code size #of qwords
        xor     eax,eax                 ; result of checksum
checksum_1:
        add     eax,cs:[si]
        add     si,08h
        add     eax,cs:[si-4]
        loop    checksum_1
        jnz     $               ;如果 checksum 查核错误，立刻陷入无限执行
                                ; 循环(由于已经 cli，此时系统形同死机)

        check_point_ini 0d1h    ; 送出 POST 检查码 0D1h 到 Port 80h
;-----------------------------------------;
; KEYBOARD CONTROLLER TEST  ;测试键盘控制器(KBC)
;-----------------------------------------;
        in      al,64h
        and     al,04h          ; sys bit (系统位) 未改变
        mov     ah,al           ; save sys bit
        mov     al,0aah         ; BAT command AA
        out     64h,al
kb_2:
        jcxz $+2
        in      al,64h
        test    al,1
        jz      kb_2
        in      al,60h          ; BAT result 55
        mov     al,60h
        out     64h,al          ; command byte

kb_3:
```

```
        jcxz    $+2
        in      al,64h
        test    al,2
        jnz     kb_3
        mov     al,ah
        or      al,61h          ;keyboard enable, mouse disable
        out     60h,al          ;键盘控制器启动，但先关掉 PS/2 鼠标功能
kb_4:
        jcxz $+2
        in      al,64h
        test    al,2
        jnz     kb_4
;---------------------------------;
;   Check for Wake Up             ;
;---------------------------------;
;(CORE0071)>
;       extrn  u_wake_up:near
        extern u_wake_up(u_wake_up_end):near
;<(CORE0071)
        public u_wake_up_end
        jmp   u_wake_up
u_wake_up_end:
;-----------------------------------------;
; Save CPU-ID in CMOS 35,36               ;
;-----------------------------------------;
        mov     edx,ebp         ;
cmos_not_good_yet:
        mov     al,0b5h         ;存取CMOS索引地址35h 的内容并且关掉NMI
        out     70h,al          ;bit7=1 off NMI ,=0 turn on NMI
        io_delay
        mov     al,dl
        out     71h,al
        io_delay
        mov     al,0b5h
        out     70h,al
```



```
        io_delay
        in      al,71h
        io_delay
        cmp     al,dl
        jnz     cmos_not_good_yet
        mov     al,0b6h
        out     70h,al
        io_delay
        mov     al,dh
        out     71h,al
;--------------------------------  ; 在本程序段落之前，主板上的 DRAM 尚未初
; START MEMORY REFRESH            ; 使化，尚不能使用，故此时还无法动用堆栈
;--------------------------------  ; (Stack)
;--------------------------------  ;
; CHIPSET REGISTER                ; 以jmp_si宏调用go_to_flat_mode，返回时设定
; PROGRAMMING                     ;
;--------------------------------  ;
jmp_si go_to_flat_mode            ; DS=ES=0(4GB 寻址模式)，并打开 GateA20 开关

        check_point_ini 0d3h      ; 送出 POST 检查码 0D3 到 Port 80h

jmp   u_cp_init                   ; 跳到外部MEMORY.ASM中的u_cp_init
u_cp_init_end:                    ; 从MEMORY.ASM返回后，DRAM已经可
                                  ; 正式运作
        check_point_ini 0d4h      ; 送出 POST 检查码 0D4h 到 Port 80h

;--------------------------------  ;
; 512 KB MEMORY TEST              ; 以下这段程序代码是检查主存储器 0～512KB
;--------------------------------  ; 的区域
        xor     ax,ax
        mov     es,ax
        mov     bx,es:word ptr [472h]   ;取得0000:0472h处的地址内容存入bx
        mov     eax,5aa5a55ah           ; 测试的32位模板(test pattern)
        mov     edx,eax
memory_test:
        xor     edi,edi                 ; EDI=00000000h
```

```
        mov     ecx,4000h*8   ; 512kb memory = 64k*8 dword
        db      67h
        rep     stosd         ;从000000h～07FFFFh存入5A A5 A5 5A
        xor     edi,edi       ; EDI=00000000h
        mov     ecx,4000h*8   ; 512kb memory = 64k*8 dword
        db      67h
        repz    scasd         ; 从000000h～07FFFFh比较是否为5A A5 A5 5A
        jnz     $             ; 比完512KB 都没有一组dword 符合，表示DRAM
                              ;功能异常(或没插DRAM)立刻陷入无穷循环(当机)
        xor     eax,edx
        jz      memory_test   ; 中间有一组dword 不合则重新跳回检查
        mov     es:word ptr [472h],bx   ; 将原先0000:472h 处标志位字节存回去
;-----------------------------------------;
;;;; jmp_si comeback_from_flat_mode       ;
;-----------------------------------------;
;  COPY 8K CODE TO SEGMENT 0              ;
;-----------------------------------------;
```

check_point_ini 0d5h ; 送出POST检查码0D5h到Port 80h

```
        ret_sp get_start_of_init_code   ; 调用get_start_of_init_code子程序。回传
                                        ; DS:DI=指向BIOS header文件头地址的下
                                        ; 一个地址，即 Boot Block 程序代码所在
                                        ; (ptr to bios header+1)
        xor     ax,ax
        mov     ds,ax
        mov     ss,ax
        mov     sp,di           ; SS:SP = 0000:start of init code
                                ; 设定DS=ES=0000h, Segment limit调为4GB
        mov     si,di           ; start offset of init code
        mov     cx,di           ; start offset of init code
        neg cx  ; CX = init code size in bytes
        shr     cx,2            ; CX = init code size #of dwords
        rep     movs es:dword ptr [di],cs:dword ptr [si]
                                ; 从F000:E000h复制8K程序代码到0000:E000h
```

```
        push    es
        push    offset cgroup:s0_00
        retf                    ;跳到0000:E0xx 原先复制的程序代码分身
s0_00:

        check_point_ini 0d6h    ;送出POS检查码0D6h到Port 80h

        call check_ctrl_home    ;execute this routine from segment 0
        pushf
        call form_rom_image     ;ESI = start address of buffer
                                ;CX = ROM size in 64k unit
        popf
        jz      cmb_03          ;若按CTRL+Home组合键，则进入软盘自我烧写模式
        call check_main_bios_checksum; 调用子程序检查BIOS checksum
        jnz     cmb_03          ;checksum查核值不正确，同样进入软盘自我烧写模式
;-------------------------------------------------;
;  TRANSFER CONTROL TO MAIN BIOS      ;
;-------------------------------------------------;

        check_point_ini 0d7h    ;送出POST检查码0D7h到Port 80h

; simulate hard reset by programming keyboard controller..
        mov     al,60h
        out     64h,al          ;keyboard command byte
        call ib_free
        mov     al,30h          ;产生硬件 reset 信号，并关掉键盘、鼠标接口
        out     60h,al
        call ib_free
        in      al,60h          ;consume data if any
;-----------------------------------;
        mov  bl,bblk_runtime_interface_module_id
        call  search_compressed_module; 找寻已压缩的模块并忽略掉
        jnz  $                  ;module not found

;AX:BX=seg:off of start of uncompressed module(AX:BX指向第一个未压缩的模块地
```

```
址) ; DI  = module size in bytes(DI为该模块的长度)
        mov     cx,di
        mov     ds,ax
        mov     si,bx
        push    ram_segment-2000h
        pop es
        xor     di,di
        push    es
        push    di              ; 将该未压缩模块直接复制、搬移到
        rep     movsb           ; (ram_segment-2000):module处
        call restore_cpuid      ; restore EDX
; EDX = CPUID at power-on
        retf    ; 此时由该未压缩模块(其实就是POST模块)程序代码
                ; 获得控制权，跳到 1000:XXXXh 处继续执行。
                ; 有完整BIOS源代码的研发厂商，可参考UUI.ASM
                ; 或 5-4、5-3 章节的说明
```

AMI SELFLASH 自我烧写机制

紧接着上述 UB.ASM 的程序代码片段，我们要介绍的是 AMI 所设计的 Emergency SELFlash 机制。以下的程序代码会取代原先由 RUNTIME.ASM 所建立 Int 19h 操作系统激活子程序。

在 BootBlock 执行的初段程序代码，一旦检测到整块 BIOS 的 Checksum 加总检查码不符合，或者用户按下 Ctrl+Home 组合键时，就会自动从放在磁盘驱动器 A:的磁盘中，寻找一个文件名为 AMIBOOT.ROM 的 ROM 文件；如果找到这个 ROM 文件，则会马上进行自我 Flash 烧写的程序。

UB.ASM 中的四组子程序依次是:

- READ_1_CLUSTER：读一组软盘的磁集 (cluster) 。
- FIND_SEC_HD_CYL_CLUSTER：将磁集转化磁头 / 磁柱 / 扇区，前述的 READ_1_CLUSTER 会调用这个子程序。
- FIND_NEXT_SECTOR_HD_CYL：寻找下一个磁集对应的磁头 / 磁柱 / 扇区 。
- ND_STORE_CLUSTER_NOS: 寻找文件对应的下一个磁集。这个子程序会再转调用上述子程序。

通过这四组短小精干的子程序，把急救用的 BIOS ROM 文件加载 RAM 之后，接下来调用 FLASH.ASM 中的 flash_program 子程序，烧写成功就重新 reset ，不成功则显示闪烁的错误信息后死机。

由于叙述焦点以及篇幅的因素，笔者必须把这四组子程序的程序代码片段的说明省略，在许多汇编语言的书中，都有类似对磁盘磁集(Cluster) 与磁头 / 磁柱 / 扇区的转换程序范例，有兴趣的读者可以自行参考。

以下假定主板最少有 512KB 左右的内存空间。

```
;------------------------------------------------ ;
;   MAIN BIOS CHECKSUM BAD or                    ;
;    <CTRL><HOME> pressed..                      ;
;   go to flash programming..                    ;
;------------------------------------------------ ;
cmb_03:
            push    es
; initialize the IO chipset (if any) for Floppy
            mov     si,offset cgroup:bootblock_fdc_table
fdc_00:
            cmp     si,offset cgroup:bootblock_fdc_table_end
```

```
        jae     fdc_end
        lods cs:word ptr [si]           ; AX = routine offset
        call    ax                      ; do not destroy SI
        jmp short fdc_00
fdc_end:
        pop es                          ; ES = 0000 with 4GB access
        push    0040h
        pop ds                          ; DS = 0040h
;;;;    jmp r_memory_0                  ; goto flash programming
;-------------------------------------------------  ;
```

```
;-------------------------------------------------  ;
;   SETUP INTERRUPT VECTORS             ; 设置 Int 13h、Int 1Eh 与 Int 0Eh 等
;-------------------------------------------------  ; 中断向量表地址
;;;;r_memory_0:

        check_point_ini 0e0h            ; 送出 POST检查码0E0h 到 Port 80h

        mov     ds:word ptr [13h],200h  ; 40:[13h]存入200h(=512KB,BIOS记
                                        ; 录主存储器字节)
        xor     ax,ax
        mov     ds:word ptr [72h],ax    ; clear CTRL-ALT-DEL(清除CTRL-ALT-
                                        ; DEL标志位字节)
        mov     es:word ptr [0eh*4],offset cgroup:int_0e
        mov     es:word ptr [0eh*4+2],ax
        mov     es:word ptr [1eh*4],offset cgroup:diskette_parameter_table
        mov     es:word ptr [1eh*4+2],ax
        mov     es:word ptr [13h*4],offset cgroup:int_40
        mov     es:word ptr [13h*4+2],ax
;-------------------------------------------------  ;
;   INITIALISE DMA CONTROLLER           ; 设置8237DMA控制器，准备读
;-------------------------------------------------  ; 磁盘数据
```

```
        out     dma_1_byte_ptr_ff_reset,al; data not important (#1)
        io_delay
        out     dma_2_byte_ptr_ff_reset,al; data not important (#2)
        mov     al,00000000b

        io_delay
        out     dma_unit_1_command_port,al; enable DMA unit (#1)
        io_delay
        out     dma_unit_2_command_port,al; enable DMA unit (#2)
        call    init_8237_dma       ; initialize DMA unit #1, #2
        mov     al,00000000b
        out     dma_unit_2_writ_req_reg,al; clear write request reg
        io_delay
        out     dma_unit_2_mask_s_r_reg,al; clear mask set/reset reg
;-------------------------------------------- ;
; 8259 INTERRUPT CONTROLLER TEST             ; 测试 8259 中断控制器并设置相关值
;-------------------------------------------- ;
intr_0:
        mov     al,00010001b                ; icw1, edge, icw4
        out     i_c_s_port,al               ; int cntlr slave port
        io_delay
        out     i_c_m_port,al               ; int cntlr master port
        io_delay
        mov     al,level_1_int              ; start vector for intr #2out
        i_s_m_port,al                       ;设置第一组 8259 中断向量号码组
        io_delay
        mov     al,level_2_int              ; start vector for intr #1out
        i_m_m_port,al                       ; 设置第一组 8259 中断向量号码组
        io_delay
        mov     al,02                       ; setup icw3, slave, level 2
        out     i_s_m_port,al
        io_delay
        mov     al,04                       ; setup icw3, master, level 2
        out     i_m_m_port,al
        io_delay
```

```
        mov     al,01                   ; setup icw4, 8086 mode,slave
        out     i_s_m_port,al
        io_delay
        out     i_m_m_port,al           ; setup icw4 for master
        io_delay
        mov     al,11111111b            ; (#2) mask value
        out     i_s_m_port,al
        io_delay
        mov     al,10111111b            ; (#1) mask value
        out     i_m_m_port,al
;-----------------------------------------------  ;
;     ENABLE INTERNAL CACHE                       ;
;-----------------------------------------------  ;
        call enable_internal_cache      ; 打开 CPU 内部的 L1/L2 Cache
;;;;    jmp int_19
;-------------------------------------------------------------------
```

以下省略

```
;-------------------------------------------------------------------;
ptr_to_source_buffer    equ  100000h  ;指定源代码缓冲区指位器地址在100000h 处
read_file_segment   equ  2000h        ; segment where BIOS file is being read
sec_1_buffer        equ  0500h        ; offset where 1st sector is being read
dir_cluster_buffer      equ  0600h    ; offset where dir sector is being read
                                      ; and where cluster#s of the file is being stored
drive_number        equ  0000h        ; 00 for A:, 01 for B:
no_of_bytes_dir_entry  equ 0020h      ;32bytes per director entry( 文件名字段有32bytes)
end_of_cluster      equ  0fffh        ; end of cluster indicator(文件最后一组
;---------------------------------- ; 磁集号码为FFFFh)
;;;;int_19:
```

```
check_point_ini 0edh      ; 送出 POST 检查码 0Edh 到 Port 80h
```

```
; F000 segment is free and can be programmed freely..
```

```
;-------------------------------------------- ;
;   FLOPPY UNIT INITIALISATION                ;
;-------------------------------------------- ;
        sti
        extrn   floppy_diskette_setup:near
        call floppy_diskette_setup              ; 调用软盘控制器(FDC)初始化程序
        jnc     dse_00                          ; ok
; (CORE0211+)>
        public  dummy_ret
        extern  atapi_setup(dummy_ret):near
        call atapi_setup                        ; ATAPI controller/device setup
                                                ; 调用ATAPI 控制器 / 安装设置子程序
        jnc short dse_00                        ; ok
; <(CORE0211+)
        mov     si,offset cgroup:floppy_cntlr_err
        jmp display_blink_halt                  ; 跳至display_blink_halt显示闪烁信息
                                                ; 并使系统暂停运行
dse_00:
;-------------------------------------------- ;
; set DS, ES for reading the 1st sector of the diskette..
        xor     ax,ax
        mov     ds,ax
        mov     es,ax
; ask user to insert the diskette in drive A:..

        check_point_ini 0eeh    ; 送出 POST 检查码 0Eeh 到 Port 80h

i19_02_00:
        mov     si,offset cgroup:flp_ins_msg    ;指定SI为提示字符串信息的偏移地址
                                                ;提醒用户将旧板BIOS ROM文件磁
                                                ;盘放入磁盘驱动器
i19_02:
        call display_on_port                    ; in drive A:

        mov     cx,8000h
```

```
        call    fixed_delay
i19_00:
; reset diskette system.. (重置磁盘驱动器)
        mov     ah,00h                  ; reset func#
        mov     dl,drive_number         ;drive A:
        int     13h                     ; reset
; read sector 1, hd 0, cyl 0..
        mov     bx,sec_1_buffer         ;  ES:BX ptr to buffer
        mov     si,bx                   ; ES:BX=ES:SI
        mov     cx,0001h                ; CH = cyl#, CL = sec#
        mov     dh,00h                  ; DH = hd#, DL = drv#
        mov     ax,0201h                ; read 1 sector(读入激活扇区)
        int     13h
        jnc     i19_01                  ; ok..
        cmp     ah,80h                  ; drive not ready ?
        jz      i19_02_00               ; yes..so try again..
        cmp     ah,06h                  ; media change ?
        jz      i19_00                  ; yes..so try again..
        pusha

        check_point_ini 0efh            ;送出POST 检查码 0Efh 到 Port 80h

        popa
i19_03:
; read error..
        mov     si,offset cgroup:read_err_msg;SI=提示字符串的偏移地址,提示
                                                读取错误jmp
        short i19_02            ;跳回19_02 处reset 软盘并重新读取磁盘
i19_01:
; ES:BX=ES:SI=DS:BX=DS:SI..ptr to content of sec 1, hd 0, cyl 0 of diskette in drive A:..
        pusha
        mov     si,offset cgroup:flp_read_msg  ; 显示磁盘读取中的信息
        call    display_on_port
        popa
```

```
; find absolute start sector# of root directory.. 找第一个根目录的绝对扇区号码
;    锣传⊖BIOS Int13h  钡      head/cylinder/sector (磁头 / 磁柱 / 扇区号码)
        mov   al,[si+10h]                    ; AL = #of FATs
        mov   ah,00h                         ; AX = #of FATs
        mul word ptr [si+16h]    ; multiply by #of sectors per FAT
                                             ; AX = #of sectors occupied by FATs
                                             ; !!!! DX destroyed by MUL instruction
        add     ax,[si+1ch]                  ; add #of hidden sectors
        add     ax,[si+0eh]                  ; add #of reserved sectors
; AX = absolute start sector# of root directory..
; AX = 根目录的绝对扇区号码(DOS INT25h/26h)

        mov     bp,ax                        ; BP = abs start sector# of root directory
        mov     ax,no_of_bytes_dir_entry     ; 32 bytes per director entry
        mul word ptr [si+11h]    ; multiply by #of entries in directory
                                             ; DX:AX = #of bytes in root directory
        div     word ptr [si+0bh]            ; divide by #of bytes per sector
        or      dx,dx
        jz      i19_04
        inc     ax
i19_04:
; AX = #of sectors occupied by root directory..AX=根目录所占用的扇区数
; BP = abs start sector# of root directory..  BP=根目录绝对扇区号码
        mov     cx,ax                        ; CX = #of sectors in root directory
        add ax,bp                            ; AX=start absolute sector# of data area
        mov     [si],ax                      ; save start absolute sector# of data area
        mov     ax,[si+0bh]                  ; #of bytes per sector
        mov     bl,no_of_bytes_dir_entry     ; 32 bytes per directory entry
        div     bl                           ; AL = #of entries in one dir sector
        mov     ah,00h
        mov     di,ax                        ; DI = #of entries in one dir sector
i19_05:
pusha
```

check_point_ini 0f0h	; 送出 POST检查码 0F0h 到 Port 80h

```
        popa
        mov     ax,bp                   ; absolute sector# of next root dir sector
        push    cx
        push    di
        mov     si,sec_1_buffer         ; DS:SI ptr to floppy BPB informn
        call find_sector_head_cyl       ; calculate cyl#, head#, sector#
        mov     bx,dir_cluster_buffer   ; ES:BX ptr to buffer
        mov     ax,0201h                ; read 1 sector
        int     13h
        jc      i19_03                  ; error..
i19_06:
; search for the pre-defined file name.. 寻找先前定义的ROM文件名(如AMIBOOT.ROM)
        push    di
        mov     di,bx                   ; ES:DI ptr to dir entry
        mov     si,offset cgroup:rom_filename; DS:SI ptr to pre-defined ROM filename
        mov     cx,000bh                ; 11 bytes in filename
        repz  cmps cs:byte ptr [si],es:byte ptr [di]; match filename in directory
        pop di
        jz      i19_07                  ; filename found..
        add     bx,no_of_bytes_dir_entry; ptr to next dir entry
        dec     di
        jnz     i19_06                  ; continue searching in this dir sector
        pop di                          ; DI = #of entries in one dir sector

        pop cx
        inc     bp                      ; abs sector# of next dir sector
        loop    i19_05                  ; goto search in next dir sector
        pusha

        check_point_ini 0f1h            ; 送出 POST 检查码 0F1h 到 Port 80h

        popa
; file not found..
        mov     si,offset cgroup:file_abs_msg
```

```
        jmp i19_02
i19_07:
        pop di
        pop cx
; file found in root dirctory..ES:BX ptr to concerned dir entry..
; store the cluster#s of the file serially in the dir_cluster_buffer..
;(找到 ROM 文件名，ES:BX= 指向相关的根目录项，并找出该文件的第一个磁集号码)
        mov     di,dir_cluster_buffer
        mov     ax,es:[bx+1ah] ; start cluster# of the file from the
                                        ; concerned directory entry
        mov     [di],ax                 ; save the start cluster number
; read the FAT sectors one by one at scratch segment..
        mov     si,sec_1_buffer
        mov     bp,[si+16h]     ; BP = #of sectors in a FAT(FAT表的扇区数)
        push    ds
        mov     bx,0040h
        mov     ds,bx                   ; DS = 0040h
        mov     bx,ds:[0013h]   ; # of 1Ks memory below 1M(取得1MB以下主存
                                ; 的 K 数
        pop ds
        shr     bx,06h          ; #of 64Ks memory below 1M
        dec     bx
        shl     bx,0ch          ; BX=640K传统主存储器以下, 所寻获的最高段地址
        mov     es,bx           ; ES = FAT 文件配置表扇区所在的段地址
                                ; going to be read
        pusha

        check_point_ini 0f2h            ; 送出POST 检查码 0F2h 到 Port 80h

        popa
        xor     bx,bx           ; 位于 fat_segment:0 区域已读入JFA扇区
        mov     dx,00h*256+drive_number ; DH,DL = head# of 1st sect, drive#
        mov     cx,0001h        ; CH,CL = cyl#, sec# of 1st sector
i19_08:
        call find_next_sector_hd_cyl            ; find sec#, hd#, cyl# of next sector
```

```
; read next FAT sector.. (读下一个FAT 扇区)
; DH,DL = head# of next FAT sector, drive#..
; CH,CL = cyl#, sector# of next FAT sector..

        mov     ax,0201h                ; 一次读一个FAT 扇区数据进来
        int     13h
        jc      i19_03_00               ; read error..
        add     bx,[si+0bh]             ; update ptr to buffer
        dec     bp
        jnz     i19_08                  ; read all FAT sectors

; all FAT sectors are read at ES:0000..
; DS:SI ptr to BPB information..
; DS:DI ptr to cluster# storage area..

        mov     ax,[di]                 ; get start cluster# of the file
        inc     di
        inc     di
        call  find_store_cluster_nos    ; find and store the cluster#s

; all cluster#s (WORD) of the file are stored serially
;    at DS:dir_cluster_buffer..
; DS:SI ptr to BPB information..
; read the file cluster by cluster..

;;;;    mov     bx,read_file_segment    ; segment where file will be read
;;;;    mov     es,bx
;;;;    xor  bx,bx
        mov     di,dir_cluster_buffer
        xchg    si,di

; DS:SI   ptr to cluster# storage area..
; DS:DI   ptr to BPB information..

        mov     dword ptr [di+80h],ptr_to_source_buffer; init ptr to source buffer
```

```
        mov     bp,[di+0dh]
        and bp,00ffh                    ; BP = #of sectors per cluster
        pusha

        check_point_ini 0f3h            ; 送出 POST 检查码 0F3h 到 Port 80h

        popa
i19_10:
        lodsw                           ; get next cluster#(取得下一个磁集号码)
        cmp     ax,0ffffh               ; end of file ?
        jz      i19_09                  ; yes..
        push    read_file_segment       ; segment:0 where next cluster will be read
        pop es
        xor     bx,bx
        push    bx

        call read_1_cluster
        pop bx
        jc      i19_03_00                       ; read error
; copy the read data to the source buffer
; ES:BX = ptr to read data, BP = #of sectors read
; DS:DI = ptr to BPB information
        pushad
        mov     ax,[di+0bh]             ; #of bytes per sector
        mul     bp
        push    dx
        push    ax
        pop ecx                         ; ECX = #of bytes read
        push    di
        mov     edi,dword ptr ds:[di+80h]       ; EDI = ptr to source buffer
        movzx esi,bx
        push    es
        push    ds
        push    es
        pop ds                          ; DS:ESI = ptr to read data
```

```
        push    0000h
        pop es                                  ; ES = 0000 with 4GB access
                                                ; ES:EDI = ptr to final source buffer
        rep     movs es:byte ptr [edi],ds:byte ptr [esi]
        pop ds
        pop es
        pop si                                  ; DS:SI = ptr to BPB information
        mov     dword ptr ds:[si+80h],edi       ; update ptr to source buffer
        popad
        jmp short i19_10                        ; continue..
i19_03_00:
        jmp i19_03                              ; read error..
i19_09:
        mov     ecx,ds:dword ptr [di+80h]       ; ptr to end of file+1
        sub ecx,ptr_to_source_buffer            ; ECX = file size in bytes
        shr     ecx,15                          ; ECX = file size in unit of 32k
; BIOS ROM file is read from floppy diskette and is stored in 1m:0000 onwards
; depending on size of the file all the scratch data area which has been used
; for reading the diskette is NOT needed anymore and can NOW be re-used for
; scratch data area for somethingelse..
;---------------------------------------;
        cli
; ECX = file size in unit of 32k
; start operating on flash..
pushad

        check_point_ini 0f5h                    ; 送出POST 检查码 0F5h 到 Port 80h
        call disable_internal_cache ; disable internal cache
        popad
        call flash_program                      ; 调用 flash_program 子程序
        jc      i19_15                          ; error..display and halt
; successful..generate cpu reset..
; make sure floppy goes off after flash programming
        mov     dx,3f2h                         ; DIGITAL OUTPUT REGISTER PORT
        mov     al,0ch                          ; turn of MOTOR (关掉磁盘驱动器马达)
```

```
        out   dx,al
;-------------------------------------;
; CS:SI = ptr to message/beep codes..
; enable GateA20..
        call display_on_port    ; 调用并显示烧写成功的信息
        mov     al,60h
        out     64h,al
        call    ib_free
        mov     al,75h          ; disable kb, set sys bit (关掉键盘)
        out     60h,al
        call    ib_free
        in      al,60h
        mov     bl,cgroup:boot_block_flag; get boot block flag
        test bl,02h             ; 需要清除掉 CMOS的checksum 存储值吗?
        jz      no_csum_destroy         ;no
        mov     ax,0c08eh               ; 清除 CMOS的checksum存储值
        call cmos_data_out
        no_csum_destroy:
        test bl,01h             ; 检测目前Flash 芯片的特性是否要硬件 reset 开机
        jnz     halt            ; 需要，跳到 halt 处设置硬件 reset 信号后等待 reset
; reboot after successful flash programming
        call restore_cpuid      ; restore EDX
;;;;    jmp far F000:FFF0
        db      0eah            ; 不需要硬件 reset，直接跳至 F000:FFF0h 处重新
        dw      0fff0h          ; reinitialize & post
        dw      0f000h          ; 此处以 db，dw 定位字节来仿真 opcode
        halt:
        mov     al,0fh
        out     61h,al
        jmp short halt
i19_15:
        jmp display_blink_halt
;-------------------------------------;
..以下UREAD_1_CLUSTER、FIND_SEC_HD_CYL_CLUSTER、FIND_NEXT_SECTOR_HD_CYL
..以及 FIND_STORE_CLUSTER_NOS 四组子程序片段说明省略 .....
```

```
;-------------------------------------------------------------------------;
        public _UB_ENDS
_UB_ENDS    label  byte          ;注明这里为UB的结束标记(_UB_ENDS)
_text       ends
        end
```

至于 0D8h、0D9h 等 Post Code 检查点，可以在 UUI.ASM(本书光盘并没有附) 中找到, 对于手边有比较完备 AMI BIOS 源代码的主板研发人员而言，可以紧接着上述 UB.ASM 的说明，来做进一步的研读。

5-4 BIOS 程序源代码浏览(二)——Flashpart (快速存取存储器)相关程序代码分析

文件名	FLASH.ASM
路径	\627.10\SIS530\SYS\FLASH.ASM┤..\..\SYS\FLASH.ASM

```
        page    ,132
        title HOOKS for BOOT BLOCK BIOS, SYSTEM BIOS, PnP Flash NVRAM
;-------------------------------------------------------------------
```

以下省略

```
;-------------------------------------------------------------------------------- ;
;批注翻译: 这段程序代码将会与Boot Block程序代码与正常Runtime程序代码连结 。;
;          在这段程序代码内任何变量、宏定义尽量先定义好，不能做外部调用;
;..... 以下省略(请看原Flash.asm英文说明) .....;
;-------------------------------------------------------------------------------- ;
```

```
        include makeflag.equ
        include mbiosmac.mac
;-------------------------------------------------------------------------;
        extrn  pgm_kbc:near
        extrn  delay_50ms:near
;-------------------------------------------------------------------------;
```

以下省略

```
;-------------------------------------------------------------------------;
cgroup  group  _text
_text   segment word  public 'CODE'
        assume  cs:cgroup
.486p
;(CORE0071+)>
;-------------------------------------------------------------------------------------;
; GET_ROM_IMAGE (取得ROM 程序影像文件所存在的内存地址)                                ;
;-------------------------------------------------------------------------------------;
; 此子程序用来取得ROM 监控数据所存在内存地址(EDI)，同时计算它有几                    ;
; 个64KB 的区块(保存到CX 寄存器)                                                     ;
    ..... 以下省略 (请看原Flash.asm英文说明) .....
;-------------------------------------------------------------------------------------;
        public  get_rom_image       ;声明get_rom_image为公用性质，可供别的
                                    ;目标码(*.obj)调用并连结之用
get_rom_image:
        push    ax
        pushf
        cli
ifMKF_256K_BIOS
        mov     ah,040h             ;
        call read_sio_byte          ;
        or      al,03h              ;
        call write_sio_byte         ;   请参考第六章关于SIS530芯片组
        mov     esi,0fffc0000h      ;   对于256KB FlashEPROM的控制方式
        mov     cx,4                ;
```

```
else
        mov     ah,040h             ;
        call read_sio_byte          ;
        or      al,02h              ;   请参考第六章关于SIS530芯片组
        call write_sio_byte         ;   对于256KB FlashEPROM的控制方式
        mov     esi,0fffe0000h      ;
        mov     cx,2                ;
endif
        popf
        pop ax
        clc
        ret
;-------------------------------------------------------------------------------;
;MAP_PHYSICAL_ROM_TO_SYSTEM_ADDRESS(实体ROM地址转成系统地址的子程序);
;-------------------------------------------------------------------------------;
;此子程序负责将ROM的32位实体地址，依芯片组的支持特性转成系统可接受;
; 内存地址也许32位4GB寻址，也许低于1MB的DOS模式，视条件而定。    ;
..... 以下省略(请看原Flash.asm英文说明) .....
;-------------------------------------------------------------------------------;
        public  map_physical_rom_to_system_address
map_physical_rom_to_system_address:
        or      edi,0fff00000h ;第20～第31地址位依原输入值保留
        clc
        ret
;-------------------------------------------------------------------------------;
; METHOD_MAPPING_PHYSICAL_ROM_TO_SYSTEM_ADDRESS                                 ;
;-------------------------------------------------------------------------------;
;此子程序保存ROM真正物理内存地址                                                ;
..... 以下省略(请看原Flash.asm英文说明) .....
;-------------------------------------------------------------------------------;
        public  method_mapping_physical_rom_to_system_address
method_mapping_physical_rom_to_system_address:
        clc                                 ;NC, mapped below 4GB

        Ret
```

```
; <(CORE0071+)
;-------------------------------------------------------------------------------- ;
; E000_READ_ROM_WRITE_X                                                          ;
;-------------------------------------------------------------------------------- ;
; 这段子程序激活 E000 段 ROM 对应，以及 ROMCS 设置，
; 可以读到在E000段译码的附加卡地址
;          ..... 以下省略 (请看原Flash.asm英文说明) .....
;--------------------------------------------------------------------------------;
        public  e000_read_rom_write_x
e000_read_rom_write_x:
        pushf
        push    ax
        cli

        mov     ah,040h
        call read_sio_byte
ifMKF_256K_BIOS
        or      al,03h
else
        or      al,02h
endif
        call write_sio_byte

        mov     ah,74h
        call read_pci_byte
        and al,NOT 88h                  ; read disable
        call write_pci_byte

        mov     ah,75h
        call read_pci_byte
        and al,NOT 88h                  ; read disable
        jmp short e_f_common
;-------------------------------------------------------------------------------- ;
; E000_READ_X_WRITE_RAM                                                          ;
;-------------------------------------------------------------------------------- ;
```

```
;这段子程序设置E000段位置区域做Shadow RAM对映，同时设置成可写入状态  ;
..... 以下省略(请看原Flash.asm英文说明) .....
;-------------------------------------------------------------------------------------;
        public e000_read_x_write_ram
e000_read_x_write_ram:
        pushf
        push    ax
        cli

        mov     ah,74h
        call    read_pci_byte
        or      al,0aah                 ;read/write enable

        call    write_pci_byte

        mov     ah,75h
        call    read_pci_byte
        or      al,0aah                 ;read/write enable
        jmp short e_f_common
;-------------------------------------------------------------------------------------;
; E000_READ_RAM_WRITE_ROM                                                             ;
;-------------------------------------------------------------------------------------;
;  此子程序设置E000段内存区域Shadow RAM对映且为只读状态                               ;
..... 以下省略(请看原Flash.asm英文说明).....
;-------------------------------------------------------------------------------------;
        public  e000_read_ram_write_rom
e000_read_ram_write_rom:
        pushf
        push    ax
        cli

        mov     ah,74h                  ;read enbale/write disable
        call read_pci_byte
        and al,NOT 22h
```

```
        or      al,088h
;       wbinvd
        call write_pci_byte

        mov     ah,75h                  ; read enbale/write disable
        call read_pci_byte
        and al,NOT 22h
        or      al,088h
        jmp short e_f_common
;-------------------------------------------------------------------------------;
; F000_READ_ROM_WRITE_X                                                         ;
;-------------------------------------------------------------------------------;
; 此子程序设置F000段内存区域做Shadow RAM对映                                    ;
..... 以下省略(请看原Flash.asm英文说明) .....
;-------------------------------------------------------------------------------;
        public  f000_read_rom_write_x
f000_read_rom_write_x:
        pushf
        push    ax
        cli

        mov     ah,76h                  ; read disable
        call read_pci_byte
        and al,not 80h
        jmp short e_f_common
;-------------------------------------------------------------------------------;
; F000_READ_X_WRITE_RAM                                                         ;
;-------------------------------------------------------------------------------;
; 此子程序设定F000段Shadow RAM可以写入数据                                      ;
..... 以下省略(请看原Flash.asm英文说明) .....
;-------------------------------------------------------------------------------;
        public  f000_read_x_write_ram
f000_read_x_write_ram:                  ; F000 shadow ram write enabled
        pushf
```

```
        push    ax
        cli

        mov     ah,76h                          ; write enable
        call read_pci_byte
        or      al,0a0h
        jmp short e_f_common
;-------------------------------------------------------------------------------  ;
; F000_READ_RAM_WRITE_ROM                                                        ;
;-------------------------------------------------------------------------------  ;
; 此子程序设置 F000 段内存区域做 Shadow RAM 对映，且设置为只读模式               ;
..... 以下省略 (请看原Flash.asm英文说明) ..
;-------------------------------------------------------------------------------  ;
        public  f000_read_ram_write_rom
f000_read_ram_write_rom:                        ; F000 shadow ram read only
        pushf
        push    ax
        cli
        mov     ah,76h                          ; read enable/write disable
        call read_pci_byte
        or      al,80h
        and al,not 20h

e_f_common:
                                                ; wbinvd
        call write_pci_byte
                                                ; wbinvd
        pop ax
        popf
        ret

;-------------------------------------------------------------------------------  ;
; MAKE_VPP_HIGH; Make_vpp_high子程序片段已移至OEMFLASH.ASM                       ;
;-------------------------------------------------------------------------------  ;
; MAKE_VPP_LOW; Make_vpp_low 子程序片段已移至OEMFLASH.ASM                        ;
```

```
;-------------------------------------------------------------------------------;
;FLASH_WRITE_ENABLE                                                             ;
;-------------------------------------------------------------------------------;
; 用途：打开 Flash EPROM 于 E000，F000 两个 64KB 段位置区域对映并开放写入      ;
..... 以下省略 (请看原Flash.asm英文说明) .....
;-------------------------------------------------------------------------------;
        extrn oem_flash_write_enable:near
        public flash_write_enable
flash_write_enable   proc  near
; write code here for chipset register programming which controls the
; flash write signal
        pushf
        push    ax

        mov     ah,045h
        call read_sio_byte
        and al,11011011b ;BIT5-->0
        or      al,00000100b ;BIT2-->1 EPROM can be flashed
        call write_sio_byte
; call any oem specific stuff
        call oem_flash_write_enable             ; in OEMFLASH.ASM
        pop ax
        popf
        ret
flash_write_enable   endp
;-------------------------------------------------------------------------------;
; FLASH_WRITE_DISABLE                                                           ;
;-------------------------------------------------------------------------------;
; 用途：关闭 Flash EPROM 于 E000，F000 两个 64KB 段位置区域并禁止写入          ;
..... 以下省略 (请看原Flash.asm英文说明) .....
        extrn   oem_flash_write_disable:near
        public  flash_write_disable
flash_write_disable  proc  near
; write code here for chipset register programming which controls the
; flash write signal(通过芯片组寄存器控制写入信号)
```

```
        pushf
        push    ax
        mov     ah,045h
        call read_sio_byte
        and al, 11011011b       ;BIT2-->0EPROMCAN'TBEFLASHEDAGAIN
        or      al, 00100000b   ;BIT5-->1 (请参考SiS530芯片组技术手册)
        call write_sio_byte
; call any oem specific stuff
        call oem_flash_write_disable        ; 调用OEMFLASH.ASM 内的
                                            ;oem_flash_write_disable
        pop ax
        popf
        ret
flash_write_disable  endp
;-------------------------------------------------------------------------------;
; GET_CHIPSET_VALUES (取得芯片组I/O索引端口数值)                                ;
;-------------------------------------------------------------------------------;
; 此子程序保存当前主板芯片组的相关索引端口 / 数据端口号码。这段程序代码只有    ;
; 在更新PnP NVRAM才会调用到，在Boot Block程序代码中不需要这段程序代码         ;
;-------------------------------------------------------------------------------;
if NOT MKF_FLASH_4GB        ; 若FLASH_4GB标志位设置，则编译下列程序片段
                            ; (BIOS.MAK 默认值 =1，故下列程序段确定会编译)
        public get_chipset_values
get_chipset_values   proc  near
        pushf
        push    bx
        push    eax
        mov   dx,0cf8h          ; 0CF8H   PCI Configuration Address Register
        mov     eax,80000074h ; output reg index 74h (DWORD boundary)
        out     dx,eax          ;此寄存器索引值必须保留, 以便从数据端口读写
io_delay
        mov     dl,0feh         ; DX = 0CFEh, data port to access Reg 76h
        in      al,dx           ; AL = 当前SiS530芯片组第76h号寄存器数值
; (current status of F000 segment)
        mov     bl,al
```

```
        pop eax
        mov     al,bl           ; AL = 当前SiS530芯片组第76h号寄存器数值
; (current status of F000 segment)
        mov   ah,al             ; AH = 要写入第76h 号寄存器的数值
        and   ah,01011111b      ; (用来关闭F000段Shadow RAM对映并设置写保护
        pop   bx                ; DX=读写SIS530 寄存器第76h号的索引端口值)
        popf
        clc                     ; Success则清除进位标志位(carry flag=0)
        ret
get_chipset_values   endp
endif
;-------------------------------------------------------------------------------;
        rwbs    =1              ;
        rwws    =1              ;
        rwdws   =0              ;
        rwbns   =1              ;
        rwwns   =0              ;
        rwdwns  =0              ; 请参考Sis530芯片组技术文件,
        srwbs   =0              ; 以及chpregrw.inc引入文件的说明
        srwws   =0              ;
        srwdws  =0              ;
        excmos  =1              ;
        apcmos =1               ;
        include chpregrw.inc    ;
;*******************************************************************************;
```

以下省略

```
;-------------------------------------------------------------------------------;
_text   ends
        end
```

5-5 BIOS 程序源代码浏览(三) —— 其他在POST 阶段参考到的程序代码分析

接下来，我们列举一些在 POST 阶段，其他设了 Post Code(检查点)的源代码片段，这次我们打开 BIOS.ASM 来举例说明。

文件名	BIOS.ASM
路径	\627.10\ASM1\BIOS.ASM┤..\..\..\ASM1\BIOS.ASM

```
        page  ,132
        title .     PROCESSOR_TIMER_PARITY_REFRESH_NMI TEST
;**************************************************************************;
;本程序片段负责刚开机阶段的Shutdown/Reset，以及移动程序代码到                ;
; Shadow RAM的工作。详细说明请参阅所附光盘中的BIOS.ASM                        ;
;                                                                          ;
..... 以下版权声明部分省略 .....
;**************************************************************************;
;-----------------------------------------;
        include   mbiosequ.equ          ; 引入BIOS整体常数定义
        include   mbiosmac.mac          ;引入BIOS程序代码宏指令
        include   cf.equ
;-----------------------------------------;
        extrn     check_point_no_stack:near
        extrn     check_point_stack:near

        public    bios_suru                     ;
        public    shutdown_0                    ; 程序代码的bios_suru   shutdown_0
        public    shutdown_77   ; shutdown_77，shutdown_88等程序进入点
    public shutdown_88          ; 声明为公用性质，可供外界调用 / 连结
    extrn  shutdown_1:near      ;shutdown after memory error
                                ; 内存测试发生错误后的当机复位
```

```
extrn   shutdown_2:near        ; v_mode exception intr error
                               ; 虚拟保护模式下例外处理程序发生错误的
                               ; 当机复位
extrn   _shutdown_33:near; shutdown after memory test
                               ; 内存测试完毕后的当机复位
extrn   shutdown_4:near        ; boot loader request shutdown
extrn   shutdown_5:near        ; user defined shutdown routine
                               ; (供使用者自定的当机处理程序使用)
extrn   _shutdown_66:near ; shutdown during memory test to display size
                               ; 在测试并显示内存容量过程的当机复位
                               ;(为了从保护模式切换到真实模式，第三章
                               ;有对此做说明)

extrn   shutdown_7:near        ; unused, (chipset memory detection)
extrn   shutdown_8:near        ; unused, (soft reset 软件复位)
extrn   shutdown_9:near        ;shutdown after block move(INT 15h/AH=88h
                               ;区块搬移)
extrn   shutdown_a:near; user defined shutdown routine
                               ; (供使用者自定的当机处理程序使用)

extrn   init_8259_80287:near          ; shut 4/5, init 8259, 80287
extrn   disable_video:near
extrn   clear_64k_memory:near
extrn   int_isr:near

extrn   _power_on_delay:byte
extrn   _software_delay:byte
extrn   ram_segment:word
extrn   flush_all_cache:near

extrn   power_on_init:near
public  power_on_init_end
extrn   hreset_clear:near
public  hreset_clear_end
extrn   sreset_clear:near
```

```
        public  sreset_clear_end
        extern  StopUsbHostController(dummy_ret):near
        extrn   dummy_ret:near
        extrn   disable_all_cache:near
        extrn   shutdown_init:near
        public  shutdown_init_end
        extrn   decompress_post_init:near
;-------------------------------------------------- ;
;   CODE  SEGMENT                                  ;
;-------------------------------------------------- ;
cgroup  group  _text
_text   segment word  public 'CODE'
        assume cs:cgroup
.486p
;-------------------------------------------------- ;
   public _BIOS_STARTS
_BIOS_STARTS  label  byte                  ;本程序代码的开始标记
;-------------------------------------------------- ;
;   FLAGS TEST (SF,ZF,PF,CF) ;                     ;
;                                                  ;
;    | S | Z | x | A | x | P | x | C |             ;
;                                                  ;
; HARD RESET OR SHUTDOWN RESET                     ;
;-------------------------------------------------- ;
;   SHUTDOWN PROCESSING                            ;
;-------------------------------------------------- ;
bios_suru:
        cli                     ; test under CLI mode
        cld                     ; ensure direction
        mov     ax,cs
        mov     ss,ax
; $$$CORE0036+>>>
        extern  wake_up(wake_up_end):near
        public  wake_up_end
        jmp wake_up             ; 检查这次reset 是否为系统苏醒时所产生?
```

```
wake_up_end:                    ;若BIOS不具备笔记本电脑必备的resume苏醒功能
                                ;则跳回到这个段落
        jmp power_on_init
power_on_init_end:
        in   al,kb_stat_port    ;读入KBC状态端口，若sys bit设为1则为软件reset
        test al,00000100b       ;否则为真正硬件产生的reset
        jnz  shut_5             ;
;-------------------------------------------------------- ;
;   VANILLA MEMORY PATCH                                  ;
;-------------------------------------------------------- ;
ifdef VANILLA_BIOS
;-------------------------------------------------------- ;
; save CPUID in cmos 35h(DL), 36h(DH)
        extrn   cmos_data_out:near
        extrn   _refresh_value:byte
        mov     ebp,edx
        mov     al,0b5h
        mov     ah,dl
        ret_sp  cmos_data_out
        mov     al,0b6h
        mov     ah,dh
        ret_sp  cmos_data_out
; start memory refresh..
        mov     al,00h          ; initialise DMA-PAGE reg.
        out     8fh,al          ;起始DMA页寄存器，准备DRAM Refresh
        io_delay
        mov     al,01010100b    ;使用CH 1作为Refresh
        out     43h,al          ; one byte count used
        io_delay
        mov     al,cgroup:_refresh_value; low byte count
        out     41h,al
        mov     cx,100h         ;400h..01/11/95
        xor     di,di
        mov     es,di
wpulse1:
```

```
        stosb
        loop    wpulse1
;---------------------------------------;
        extrn   vanilla_patch_offset:near
        cmp     cgroup:word ptr vanilla_patch_offset,0fffh
        jz      ret_off                 ; no routine
        mov     sp,offset cgroup:sp_ret_off
        jmp vanilla_patch_offset-1
        even
sp_ret_off:
        dw      offset cgroup:ret_off
        dw      0f000h
ret_off:
endif
;---------------------------------------;
        jmp hreset_clear                ; 若纯硬件reset则跳到hreset_clear
;;;;hreset_clear_end:
;;;;    jmp shutdown_0x                 ; hard reset, goto regs. test
shut_5:
        jmp sreset_clear                ; GA20 disable and other....
sreset_clear_end:
        mov     al,8fh                  ; 从CMOS(0Fh)取得当机指示位,
        out     cmos_addr_port,al       ; 并且关掉NMI(Bit7=1)
        jcxz short $+2                  ; i/o delay
        jcxz short $+2                  ; i/o delay
        in      al,cmos_data_port       ; read shutdown code
        mov     ah,00
        mov     si,ax                   ; save in (SI)
        mov     al,8fh                  ; shutdown address
        jcxz short $+2                  ; i/o delay
        out     cmos_addr_port,al
        mov     al,00                   ; clear shutdown byte
        jcxz short $+2                  ; i/o delay
        jcxz short $+2                  ; i/o delay
        out     cmos_data_port,al
```

```
        mov     ax,cs
        mov     ss,ax
        mov     ax,si                   ; restore shutdown code
        cmp     al,04h                  ; for BOOT loader shutdown
        jz      shut_1                  ; & for USER defined shutdown
        cmp     al,05h                  ; initialize 8259 (#1, #2)
        jz      shut_1                  ; and (DS), (SS), (SP), STI
        cmp     al,0ah                  ; if shut code > 10
        jbe     shut_2                  ; bypass intr. init
        jmp short shutdown_0            ; then hard reset
;-------------------------------------------------------------- ;
;   INIT 8259 for SHUTDOWN 04, 05                               ;
; 当机指示位字节 04，05 处理例程， Initialize 8259              ;
;-------------------------------------------------------------- ;
shut_1:
mov  bx,level_2_int*256+level_1_int
ret_sp  init_8259_80287     ; shut 4/5, init 8259, 80287
;-----------------------------------------------------------------------------------------------;
; INTERNAL CACHE IS ALWAYS ON ; 指CPU 核心的L1/L2 Cache(K6-III)永远打开
;-----------------------------------------------------------------------------------------------;
shut_2:
        jmp shutdown_init
shutdown_init_end:
        mov     ax,40h                  ; global data segment (ah) = 0
        mov     ds,ax                   ; setup (DS)
mov  al,00h                             ; global extra segment (ah) = 0
        mov     es,ax                   ; setup (ES)
        mov     al,30h                  ; global stack segment (ah) = 0
        mov     ss,ax                   ; setup (SS)
        mov     sp,0100h                ; setup (SP)
        shl     si,1                    ; prepare index
        jmp [si+cgroup:shut_jmp_tbl]    ; branch with intr. disabled
;-----------------------------------------------------------------------------------------------;
;   SHUTDOWN JMP TABLE  ; 根据当机位为索引值的处理程序跳跃表
;-----------------------------------------------------------------------------------------------;
```

```
        even
shut_jmp_tbl    label  word
        dw      offset cgroup:shutdown_0        ; hard reset
        dw      offset cgroup:shutdown_1        ; used for block move internal shutdown
        dw      offset cgroup:shutdown_2        ; v_mode exception intr error
        dw      offset cgroup:_shutdown_33      ; shutdown after memory test
        dw      offset cgroup:shutdown_4        ; shutdown for boot loader
        dw      offset cgroup:shutdown_5        ; shutdown (with intr. init)
        dw      offset cgroup:_shutdown_66      ; shutdown during memory test to display
                                                ; memory size
        dw      offset cgroup:shutdown_7        ;unused, (chipset memory detection)
        dw      offset cgroup:shutdown_8        ; unused, (soft reset)
        dw      offset cgroup:shutdown_9        ; shutdown after block move
        dw      offset cgroup:shutdown_a        ; shutdown (w/o intr. init)
;--------------------------------------;
hreset_clear_end:
shutdown_0x:                    ;其他当机指示位值代表硬件产生的reset
        mov   al,8dh
        out   cmos_addr_port,al             ; 关掉NMI
;--------------------------------------;
shutdown_0:                                 ; hard reset
shutdown_77:                                ; unused
shutdown_88:                                ; unused

        check_point_si 03h                  ; 送出POST 检查码0D0h 到 Port 80h

        mov     ax,cs
        mov     ss,ax
        xor     bp,bp
        mov     ds,bp                       ; set (DS) = 0
        mov     es,bp                       ; set (ES) = 0
        jmp_di  disable_video
        in      al,kb_stat_port             ; if sys_flag bit is set
        test al,00000100b                   ; then soft reset else power on
        jz      not_cnt_alt_del             ; power on
```

```
        cmp     ds:word ptr [0472h],1234h    ;检查0:[0472h]是否为1234h?
        jnz     cnt_alt_del
yes_cnt_alt_del:
        or      bp,soft_reset_bit; msb used for soft reset
        jmp short cnt_alt_del
not_cnt_alt_del:
        cmp     ds:word ptr [0472h],1234h
        jz      yes_cnt_alt_del
        or      bp,power_on_bit
cnt_alt_del:
;;;;    ret_sp  clear_64k_memory             ; clear segment 0
        mov     ax,30h
        mov     ss,ax                        ; set stack
        mov     sp,100h

        check_point  05h                     ; 送出POST检查码 0D0h 到 Port 80h

;;      call StopUsbHostController           ; disable USB host controller
;;      call disable_all_cache               ; disable all cache
        call disable_all_cache               ;disable all cache(关掉所有主板上的寄存)
        call StopUsbHostController           ;disable USB host controller(关闭USB
                                             ;控制器)
        mov     ax,cs
        mov     ss,ax
        ret_sp  clear_64k_memory             ; 调用 clear_64k_memory清除最前面
                                             ;64K 内存
        mov     ax,30h
        mov     ss,ax                        ; set stack
        mov     sp,100h

        check_point  06h                     ; 送出 POST检查码 0D0h 到 Port 80h

        call decompress_post_init            ; 解压BIOS 中的 POST 与P INIT模块
        extrn   _bios_to_rm:near
        jmp _bios_to_rm                      ;跳到CF.ASM中的_bios_to_rm子程序处
```

```
;-------------------------------------;
;*************************************************************************
;..... 以下版权声明部分省略(请看原 BIOS.ASM) .....
;*************************************************************************

;-------------------------------------------------------------------------------- ;
; COPY_CONTROL_TO_RAM                                                             ;
; 这个子程序 copy 64KB 程序代码到特定 RAM 的段内存地址，并将控制权转              ;
; 移到位于 RAM 的程序代码去执行。(详情请看原始 BIOS.ASM 的说明)                   ;
;-------------------------------------------------------------------------------- ;
        extrn  copy_64k_memory:near
        public copy_control_to_ram
        copy_control_to_ram   proc   near
        push    es
        push    ds
        mov     es,cgroup:ram_segment       ;target segment(=ram_segment=3000h)
        push    cs
        pop ds                      ; source segment
        call copy_64k_memory ; 调用 copy_64k_memory 子程序
; transfer control to ram segment..
        push    es                          ; ram_segment
        push    offset cgroup:xxx
        retf
xxx:
        pop ds
        pop es
xxx_exit:
        ret
        copy_control_to_ram   endp
;-------------------------------------------------------------------------------- ;
; COPY_TO_SHADOW (COPY 程序代码到 Shadow RAM)                                     ;
;-------------------------------------------------------------------------------- ;
; 本子程序将指定内存地址的程序代码 copy 到指定的 Shadow RAM 区域                  ;
; ES:DI = destination segment:offset (移动目的地的段地址: 偏移地址)               ;
; DS:SI = source segment (要移动的原始程序代码段地址: 偏移地址)                   ;
```

```
;--------------------------------------------------------------------------------------;
        extrn   move_bytes:near
        extrn   enable_shadow_write:near
        extrn   disable_shadow_write:near
        public  copy_to_shadow
copy_to_shadow proc  near
        pushf
        cli                             ;移动前先禁止中断信号产生
        call  enable_shadow_write       ;enable write to shadow(设置Shadow可写入)
        call  move_bytes                ;move bytes
        call  flush_all_cache           ;flush all cache(把CPU/主板寄存清除掉)
        call  disable_shadow_write      ;enable write to shadow
        popf
        ret
copy_to_shadow endp
;--------------------------------------------------------------------------------------;
        extrn  e000_read_rom_write_x:near    ;
        extrn  e000_read_ram_write_rom:near  ;
        extrn  e000_read_x_write_ram:near    ;   需要调用其他目标码(*.obj)
        extrn  f000_read_rom_write_x:near    ;   的子程序进入点
        extrn  f000_read_ram_write_rom:near  ;
        extrn  f000_read_x_write_ram:near    ;
        public do_f000_read_rom_write_x      ;
        public do_f000_read_ram_write_rom    ;
        public do_f000_read_x_write_ram      ;   下列子程序声明为公用性质,其
       public do_e000_read_rom_write_x       ;   他目标码(*.obj)可连结与调用
        public do_e000_read_ram_write_rom    ;
        public do_e000_read_x_write_ram      ;
;--------------------------------------------------------------------------------------;
ifdef   VANILLA_BIOS
do_f000_read_rom_write_x:
        call    f000_read_rom_write_x
        mov     al,15                        ;offset to table of fn#
        jmp short vanilla_patch
do_f000_read_ram_write_rom:
```

```
            call    f000_read_ram_write_rom
            mov     al,18                   ; offset to table of fn#
            jmp short vanilla_patch
do_f000_read_x_write_ram:
            call f000_read_x_write_ram
            mov     al,21                   ; offset to table of fn#
            jmp short vanilla_patch
do_e000_read_rom_write_x:
            call e000_read_rom_write_x
            mov     al,24                   ; offset to table of fn#
            jmp short vanilla_patch
do_e000_read_ram_write_rom:
            call e000_read_ram_write_rom
            mov     al,27                   ; offset to table of fn#
            jmp short vanilla_patch
do_e000_read_x_write_ram:
            call e000_read_x_write_ram
            mov     al,30                   ; offset to table of fn#
vanilla_patch:
            push    bx
            mov     bx,cgroup:word ptr vanilla_patch_offset
            inc     bx
            jz      dop_00                  ; no routine
            dec     bx
            cbw                             ; AX = offset to jmp table(跳越地址表偏移地址)
            add     bx,ax
            push    cs
            push    offset cgroup:dop_00
            push    cs
            push    bx
            retf
dop_00:
            pop bx
            ret
;-------------------------------------;
```

```
        else                                    ; NORMAL BIOS CODE
do_f000_read_rom_write_x:
        jmp f000_read_rom_write_x               ; enable F000 ROM,并从ROM 读数
                                                ; 据写到任意RAM区段
do_f000_read_ram_write_rom:
        jmp f000_read_ram_write_rom; 设置F000 ShadowRAM 区域只读
                                                ; (可读不可写)
do_f000_read_x_write_ram:
        jmp f000_read_x_write_ram               ; 将F000 ShadowRAM 设置为可写入,
                                                ; 并从其他区段copy程序代码或数据
do_e000_read_rom_write_x:
        jmp e000_read_rom_write_x               ; enable E000 ROM, 并从ROM 读数
                                                ; 据写到任意RAM区段
do_e000_read_ram_write_rom:
        jmp e000_read_ram_write_rom;设定E000 ShadowRAM 区域只读
                                                ; (可读不可写)
do_e000_read_x_write_ram:
        jmp e000_read_x_write_ram               ; 将E000 ShadowRAM 设置为可写入,
                                                ; 并从其他区段
; copy程序代码或数据
endif
;--------------------------------------------------------;
;********************************************************************
;..... 以下版权声明部分省略 (请看原 BIOS.ASM) .....
;********************************************************************
;---------------------------------- ;
public _BIOS_ENDS
_BIOS_ENDS   label byte         ; 本程序代码 / 模块的结束标记(_BIOS_ENDS)
;---------------------------------- ;
_text ends
        end
```

其他关于 POST 以及 Runtime 的相关程序代码 / 模块

除了前面介绍过的 UB.ASM、FLASH.ASM 以及上述的 BIOS.ASM 之外，还有如 UP.ASM、UUI.ASM、RM.ASM、VM.ASM、GS1.ASM/NPOST.ASM、GS2.INC 等等，都是整个 POST 过程中，Post Card/Code 从一开机的 D0，一路切换显示到 00(Boot)阶段会扫过的程序代码片段。

由于篇幅的因素，我们不在这里作一行一行的叙述，有兴趣深究的读者，就请参考着 5-2 的执行流程图，从光盘片中的各目录中仔细搜寻、细细品读吧！

如果读者您有幸还是属于 OEM/ 主板研发团队中的一员，也许你会发现手边的 BIOS 的.ASM 文件，其实还会比作者所拥有的还要丰富；或者在 Post Code 、文件名甚至目录结构上会有些差异，其实这点差异是很正常的：本来 BIOS 厂商在跟 OEM 系统 / 主板厂商合作时，会根据设计产品、零组件的不同，而从该公司的源代码数据库中给予不同的解决方案。

本书所附的源代码光盘，也许不如 OEM/ 主板合作厂商所付费购买的 BIOS 源代码那样齐全，但无论如何，本书所附的源代码光盘内容，绝对可以生成一份可完整烧写、执行的 SiS530 版本的 AMI BIOS，对于读者的实作绝对不会有任何影响，这点是毋庸置疑的。

CHAPTER

6

BIOS 平台移植的关键技术——特定平台修改与关键模块/源代码的介绍

如果把整个系统的运行法则做大致上的轮廓划分，那么BIOS核心层程序代码，如同是中央政府的法规，那么进入芯片层这边的细雕琢磨，就好比是地方政府针对地方民情需要所制定的地方施行细则。本章的重点，就是要让这套AMI BIOS能够入乡随俗，按照我们的设计/实作平台的芯片组，外围I/O芯片的差异，来细细雕琢真正可以完美搭配的BIOS监控程序。

6-1 掌握芯片组、外围 I/O 芯片的相关规格

计算机平台的结构差异，除了CPU规格/核心结构有所不同外，构成主板的系统芯片组(Chipset)也是关键：即使是在同一代CPU平台所开发的BIOS，由于每套芯片组不光是规格、功能的差异，从寄存器、I/O端口号码上的差别，到DRAM Bank数的不同；甚至连initialize(初始化)外围装置的顺序与做法都有所不同，所以真正在做BIOS移植作业时，最常变动的部分，反而是芯片层以及OEM层这两个部分为主。

一般说来，芯片组厂商开发一套系统芯片组时，也会有相关的软件人力去研发这个示范平台所需要的BIOS程序代码，并且编写这套芯片组的BIOS Programming & Design Guide (BIOS程序设计建议指南)，BIOS厂商则从这套指南去获取相关信息，或者直接从芯片组厂商取得示范性的源代码，把手边的BIOS源代码porting到这套芯片组。当然有时候更是双方直接合作，双方一同开发针对这套芯片组的BIOS监控程序，以主板/OEM业务推广互利为合作原则。

以本书所移植实现的 SiS530 芯片组结构的主板，北桥芯片是使用 SiS530，南桥芯片则使用 SiS5595 芯片，我们先将 SiS530芯片组规格大致介绍如下：

(A) SiS530 北桥芯片：

- 支持 321pin Socket 7 插槽结构的 Intel Pentium，Pentium MMX 处理器，以及兼容于 Socket 7 结构的 AMD K6，K6-2，K6- III 以及 Cyrix 6x86，6x86MX / M II 处理器。
- 整合 SiS6326 2D/3D 图形显示芯片电路、内存控制电路、PCI/AGP 总线控制器于单一芯片内。最大图形分辨率 1600x1200@85Hz 。
- 支持 AGP 1x/2x, PCI 2.2 规格，四组 PCI 插槽 / 外围的 PCI Mastering 能力。
- 支持 100/100MHz，95/95MHz，83/83MHz，75/75MHz，66/66MHz 的 CPU/DRAM 频率同步操作模式，以及 100/75MHz，95/75MHz，83/66MHz，66/100MHz 与 66/83MHz 的异步操作模式。
- 支持多达 2MB 的 Pipeline Burst Cache 为 L2 Cache (或 L3 Cache，针对 K6-III 而言)。
- 6 组列驱动信号触发，相当于三组单 / 双面内存插槽的驱动能力，最大内存容量为 1.5GB 。
- 图形芯片可采区域显示内存 (LFB，Local Frame Buffer) 或 UMA (Unified MemoryArea, 跟主板共享 / 共享)的设计。当采取 UMA 设计时，固定占用 Bank0 的内存空间。
- 内建 IDE 硬盘控制接口，支持到 ATA/66 传输规格。

(B) 南桥芯片 SiS5595：

- PCI to ISA 桥接芯片。内建键盘控制器 (KBC)、软盘控制器 (FDC)以及 CMOS/RTC 电路功能。
- 内置双连接端口的 USB 控制器，属 OHCI (Open Host Control Interface) 规范设计，符合 USB 1.1 规格。
- 内置硬件监控电路，可监测多达五组电压值，两组 3pin 接头规格的 CPUFan、SysFAN 风扇马达转速侦测，以及 CPU 温度检测等功能。
- ver 2.0 版起加入 PMU (Power Management Unit 电源管理单元)，符合 APCI 1.0 规范。

以下是我们经过硅统公司授权，取自该公司 0530ds10.pdf (SiS530 Host, PCI，3D Graphics & Memory Controller)的技术数据文件中，第 73～78 页寄存器项目的说明：

一、HOST-TO-PCI BRIDGE CONFIGURATION SPACE HEADER (HOST 至 PCI 桥接端组态文件头区)

DEVICE IDSEL	FUNCTION	NUMBER
Host to PCI bridge	AD11	0000b

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
00-01h	Vendor ID	1039h	RO
02-03h	Device ID	0530h	RO
04-05h	PCI Command Register	0005h	RO, R/W
06-07h	PCI Status Register	0210h	RO, WC
08h	Revision ID	02h	RO
09h	Programming Interface	00h	RO
0Ah	Sub-Class Code	00h	RO
0Bh	Base Class Code	06h	RO
0Ch	Cache Line Size	00h	RO
0Dh	Master Latency timer	FFh	R/W
0Eh	Header Type	80h	RO
0Fh	BIST	00h	RO
10-13h	Graphic Window Base Address	00000000	RO, R/W
14-33h	Reserved	00h	
34h	Capability Pointer	C0h	RO

二、HOST/L2 CACHE/DRAM (HOST 与 On Board L2 Cache，DRAM 控制器部分)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
50h	NA# Control	00h	R/W
51h	L2 Cache Control Register	00h	R/W
52h	L2 Cache/DRAM Function Control	00h	R/W
53~54h	Reserved	0000h	R/W
55h	DRAM Bank open/close Control	00h	R/W

56h	Refresh Control	00h	R/W
57h	Reserved	00h	R/W
58h	DRAM Control	50h	R/W
59~5Bh	Reserved	000000	R/W
5Ch	DRAM Initialization Command	00h	R/W
5Dh	DRAM Control	00h	R/W
5Eh	DRAM Control	00h	R/W
5Fh	Reserved	00h	R/W
60h	DRAM Types of Bank 0	0000h	R/W
61h	DRAM Types of Bank 1	0000h	R/W
62h	DRAM Types of Bank 2	0000h	R/W
63h	DRAM Status Register	00h	R/W

三、CURRENT DRIVING (电流强度驱动设置)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
64h	DRAM Signals Driving Current Control	00h	R/W
65h	PCI Signals Driving Current Control	00h	R/W

四、POWER MANAGEMENT (电源管理)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
68h	SMRAM Access Control	00h	R/W
69h	System Event Monitor Control for Power Management	00h	R/W
6A~6Bh	ACPI I/O Base Address Register	0000h	R/W
6Ch	DRAM Self-Refresh Control for Power Management	00h	R/W

五、SHADOW RAM & NON-CACHEABLE AREA (设置 Shadow RAM 与非寄存区域)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
70~75h	Shadow RAM Register	00h	R/W
76h	BIOS Shadow Attribute	00h	R/W
77h	Non-cacheable Area Characteristics	00h	R/W
78~79h	Allocation of Non-cacheable area I	0000h	R/W
7A~7Bh	Allocation of Non-cacheable area II	0000h	R/W

六、HOST BRIDGE & PCI ARBITER (HOST 桥接端与 PCI 仲裁设置)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
80h	Target Bridge to DRAM Characteristics	00h	R/W
81h	PCI Target Bridge Characteristics	00h	R/W
82h	PCI Target Bridge Bus Characteristics	00h	R/W
83h	CPU to PCI Characteristics	00h	R/W
84~85h	PCI Grant Timer	FFFFh	R/W
86h	CPU Idle Timer for PCI	FFh	R/W
87h	CPU to PCI Characteristics and Arbitration option	00h	R/W
88~89h	PCI Discard Timer	0000h	R/W

七、CLOCK CONTROL (时钟设置)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
8Ah	TAGWE# Signal Control	5Ah	R/W
8Bh	Reserved	00h	R/W
8Ch	SDRCLK/SDWCLK Control	2Ah	R/W
8Dh	Reserved	00h	R/W

八 、DISPLAY STATUS(显示组态设置)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
8Eh	Integrated VGA Enable & Monochrome Device Adapter Existence Register	00h	R/W

九 、GART AND PAGE TABLE CACHE (GART 图形地址转换表与页转换寄存表格设置)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
90~93h	GART Base Address for Re-mapping	00000000	R/W
94h	Graphic Window Control	00h	R/W
95~96h	Reserved	0	R/W
97h	Page Table Cache Control	00h	R/W
98h	Page Table Cache Invalidation Control	00h	R/W

十 、DRAM PRIORITY TIMER (DRAM 优先权计时器)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
A0~A1h	CPU/PCI-GUI Privilege Timer	0000h	R/W
A2h	CPU/PCI-GUI Privilege Timer Control	00h	R/W
A3h	VGA Grant Timer	00h	R/W

十一、A.G.P. AND HOST BRIDGE (AGP 与 HOST 桥接端)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
C0~C3h	A.G.P. Capability Identify Register	00200002h	RO
C4~C7h	A.G.P. Status Register	F000203h	RO
C8~CBh	A.G.P. Command Register	00000000h	R/W

十二、DEVICE 2 (VIRTUAL PCI-TO-PCI BRIDGE) → 第二组虚拟 PCI 对PCI 桥接装置

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
00-01h	Vendor ID	1039h	RO
02-03h	Device ID	0001h	RO
04-05h	PCI Command Register	0000h	RO, R/W
06-07h	PCI Status Register	0000h	RO
08h	Revision ID	00h	RO
09h	Programming Interface	00h	RO
0Ah	Sub-Class Code	04h	RO
0Bh	Base Class Code	06h	RO
0Ch	Cache Line Size	00h	RO
0Dh	Master Latency timer	00h	RO
0Eh	Header Type	01h	RO
0Fh	BIST	00h	RO
19h	Secondary Bus Number	00h	R/W
1Ah	Subordinate Bus Number	00h	R/W
1Ch	I/O Base	F0h	R/W, RO

1Dh	I/O Limit	00h	R/W, RO
1Eh	Secondary PCI-PCI Status	0000h	WC, RO
20~21h	Non-prefetchable Memory Base Address	FFF0h	R/W, RO
22~23h	Non-prefetchable Memory Limit Address	0000h	R/W, RO
24~25h	Prefetchable Memory Base Address	FFF0h	R/W, RO
26~27h	Prefetchable Memory Limit Address	0000h	R/W, RO
28~3Dh	Reserved	0000h	RO
3Eh	PCI to PCI Bridge Control	0000h	RW, RO

十三、PCI IDE DEVICE (PCI IDE 控制器的设置)

REGISTER ADDRESS	REGISTER NAME	DEFAULT VALUE	ACCESS TYPE
00-01h	Vendor ID	1039h	RO
02-03h	Device ID	5513h	RO
04-05h	PCI Command Register	0000h	RO, R/W
06-07h	PCI Status Register	0000h	RO
08h	Revision ID	D0h	RO
09h	Programming Interface	00h	RO
0Ah	Sub-Class Code	01h	RO
0Bh	Base Class Code	01h	RO
0Ch	Cache Line Size	00h	RO
0Dh	Master Latency timer	00h	RO
0Eh	Header Type	80h	RO
0Fh	BIST	00h	RO
10~13h	Primary Channel Command Block Base Address Register	00000000	R/W

14~17h	Primary Channel Control Block Base Address Register	00000000	R/W
18~1Bh	Secondary Channel Command Block Base Address Register	00000000	R/W
1C~1Fh	Secondary Channel Control Block Base Address Register	00000000	R/W
20~23h	Bus Master IDE Control Register Base Address	00000000	R/W
24~2Bh	Reserved		
2C~2Dh	Subsystem Vendor ID	0000h	R/W
2E~2Fh	Subsystem ID	0000h	R/W
30~33h	Expansion ROM Base Address	00000000	R/W
40h	IDE Primary Channel/Master Drive Data Recovery Time	00h	R/W
41h	IDE Primary Channel/Master Drive Data Active Time	00h	R/W
42h	IDE Primary Channel/Slave Drive Data Recovery Time	00h	R/W
43h	IDE Primary Channel/Slave Drive Data Active Time	00h	R/W
44h	IDE Secondary Channel/Master Drive Data Recovery Time	00h	R/W
45h	IDE Secondary Channel/Master Drive Data Active Time	00h	R/W
46h	IDE Secondary Channel/Slave Drive Data Recovery Time	00h	R/W
47h	IDE Secondary Channel/Slave Drive Data Recovery Time	00h	R/W
48h	IDE Status Register	00h	R/W
49h	Reserved		

4Ah	IDE General Control Register 0	00h	R/W
4Bh	IDE General Control Register 1	00h	R/W
4C~4Dh	Prefetch Count of Primary Channel	FFFFh	R/W
4E~4Fh	Prefetch Count of Secondary Channel	FFFFh	R/W
50~51h	IDE minimum accessed time register	0000h	R/W
52h	IDE Miscellaneous Control Register	00h	R/W

Super I/O 芯片的搭配

一般在选用有内置 KBC 以及 RTC 的 Super I/O 芯片时，如果搭配到也有内置 KBC 以及 RTC 的芯片组(例如本书所示范的 SIS530)，在 KBC、RTC 这部分功能就重叠，两边的 KBC/RTC 都打开时，会导致冲突的状况。所以有具备 RTC、KBC 电路功能的 Super I/O 以及系统芯片，一开机的默认值都是关闭的；BIOS 在设计、编写或编译时，必须决定使用系统芯片组的 KBC、RTC，还是 Super I/O 芯片的 KBC、RTC，通常这个项目可以在 IO.MAK 维护文件中指定，前面第 4 章已经介绍过。

通常 Super I/O 芯片的 FDC，会跟一些也有内置 FDC 的南桥芯片相冲突，万一厂商采用就是这一类的芯片组合，就必须从 BIOS 这边去排解纠纷，做法可能是舍弃其中一组的 FDC 电路不用，在 POST 阶段并不打开(not enable)。当然真正取决的关键，往往是当时在 Lay 主板时，是用 Super I/O 的 FDC 线路会比较好设计，还是用南桥芯片的 FDC 线路会比较好设计来决定。

当然，要是所采用的系统芯片组已经都具备 Super I/O 的一切功能，Super I/O 这部分芯片形号、规格上的差异，就可以直接忽略掉，对于 BIOS 的规划上还可以更简化一些。

由于SiS5595已经内置键盘控制器(KBC)以及RTC (CMOS/RTC，CMOS计时芯片)，许多以SiS530芯片结构规划的主板，公板上建议采用联阳ITE8661F Super I/O芯片，将规格简介如下：

(C) ITE8661F Super I/O芯片

① 支持PnP 1.0a规格的独立PnP芯片装置，I/O，IRQ，DMA可动态调配，具备七组可调式IRQ号码、四组可选择DMA以及IRQ Sharing共享功能。

② 标准Super I/O功能，内置one FDC、two COM port(with IR port)、1 printer port、1 Game & MIDI I/O以及13组GP I/O信号线。

③ 磁盘控制器——FDC支持两部360K/720K/1.2M/1.44M/2.88M规格的磁盘驱动器

④ 多传输模式的并列端口(Multi-mode Parallel Port)，支持Standard Mode单/双向传输，Enhanced Mode(EPP V1.7&V1.9)以及高速ECP传输模式。

⑤ 内置两组16C550 UART兼容的串行端口传输接口。并提供Infrared Inteface红外线传输接口，支持HPSIR以及ASKIR等IR传输规格。

⑥ 100-pin QFP封装

由于ITE8661F正好是没有具备KBC以及RTC的设计，虽然它有FDC(不然怎么叫SuperI/O呢?)，而SiS530芯片组中的SiS5595南桥恰好没有FDC，做主板平台规划时，BIOS这部分的编译规划、维护上可以比较简单一些。

OEM 厂商的特定支持

有时厂商会有自己的平台规划，用不同的零组件，为自己的产品添加特色，或者在市场上做功能区隔。用不同的 Super I/O 芯片或南桥芯片，不见得都要照公板的设计，这时BIOS 要加入的模块又得有所变动。

甚至有些中高阶功能 / 价位的主板，常会有些附加芯片，例如另外加一组 SCSI 控制器芯片、IDE/RAID 控制芯片、10/100Mbps Ethernet 等等，这时候就得另外考虑加入这些额外芯片的线路规划，好好地做所谓的 PCI IRQ Routing，同时加入这些芯片的 ROM 控制码模块。

有些时候为了降低成本，例如用了那一家的 PCB 版，由于电流阻抗的关系，光是在 OEM 或芯片层这边修改源代码还不够，必须要深达核心层去修改一开始的驱动程序码 (如前面介绍过的 UB.ASM)，一开机就要赶快 Initial Chipset & Register，甚至驱动电流参数值等，都还要有不同的设计。这种“因成本变动”的特殊修改版，固然是因应厂商的需要而特别制造，既然厂商有需要，BIOS 厂商以及研发支持人员自然要全力以赴。

总之，要移植 AMI BIOS 到任何主板上，您起码要对北桥芯片、南桥芯片以及 Super I/O 芯片规格要完全地掌握住，而且对于这些芯片之间，有没有哪些功能 / 规格的电路，彼此兼有功能重复甚至重叠的情形，您必须做一个取舍，而取舍 (enable/disable)的顺序，也直接影响到您的 BIOS 在 OEM 层的细部规划的程序。您对硬件了解的越多，越能够精准的取舍，用最少的设计成本，达到最大的效益。

现在，我们掌握了 SiS530 主板上，有关 SIS530 北桥芯片、SiS5595 南桥芯片，以及 ITE8661F Super I/O 芯片三大组件的规格，也对它们有一定程度的认识。现在就看看从核心层程序代码下放到地方(芯片层与 OEM 层)时，该如何“因地制宜”了。

6-2 移植到特定芯片组要修改的源代码

当核心层程序要转移到芯片层时，首先要做的是：

① 为该芯片组造一连串目录。(以下 newchipset 为芯片组代号，以斜体字表示)

CD\627.10
MD *newchipset*
MD *newchipset*\ACPI
MD *newchipset*\CORE.MOD
MD *newchipset*\CPU
MD *newchipset*\DIM
MD *newchipset*\DIM\NDIM
MD *newchipset*\GPC
MD *newchipset*\GPC\NGPC
MD *newchipset*\INC
MD *newchipset*\INCLUDE.ASL
MD *newchipset*\IO
MD *newchipset*\MP
MD *newchipset*\MP\NMP
MD *newchipset*\P6PATCH
MD *newchipset*\ROM
MD *newchipset*\SYS
MD *newchipset*\SYS\NSYS
MD *newchipset*\USB
MD *newchipset*\USB\NUSB

若读者、BIOS 维护者是从 BIOS 厂商取得此 newchipset 的专属版本，则上述各子目录多半已经建立妥当，则第一、第四以及第六步骤可以省略。

② 以下再造个 OEM 目录层，以及厂商层 (如 DEMO)：

```
CD\627.10\newchipset   ←— 要 porting 的芯片组目录代号
MD OEM
MD OEM\DEMO (或其他厂商名称)
```

③ 若想对该项 OEM 计划预留扩充、更改的空间，则可执行下列指令：

```
CD\627.10\newchipset\OEM\DEMO
MD OEMPORT   ←— 造出对映于芯片层 newchipset 目录底下 SYS 目录
MD OEMPORT\ACPI.ASL   ←— 造出对映于芯片层 newchipset 目录底下的 IO 目录
MD OEMIO.MOD          ←— 如果 OEM 厂商需要修改 ACPI 相关监控程序的话
MD OEMCORE.MOD        ←— 造出对映于芯片层 newchipset 目录底下的 CORE.MOD
                         目录，用来取代最上游核心层*.ASM 以及*.SSP 设置描
                         述文件
MD OEMCHIP.MOD        ←— 用来取代上游芯片层的*.ASM 以及*.SSP 设置描述文件
```

④ 将 \627.10\ASM1、\627.10\ASM2 目录下的源代码文件，copy 到芯片层目录下的 SYS 目录：

```
cd\627.10
xcopy asm1\*.*      newchipset\sys\
xcopy asm1\*.*      newchipset\sys\nsys\
xcopy asm2\*.*      newchipset\sys\
xcopy asm2\*.*      newchipset\sys\nsys\
xcopy cpucore\*.*   newchipset\cpu\

xcopy p6ucode\*.*   newchipset\p6patch
xcopy mp\*.*        newchipset\mp\*.* /s/e
```

⑤ 将上游核心层与芯片层的相关设置描述文件*.SSP复制到OEM厂商层底下各目录：

```
xcopy obj\core.ssp          newchipset\oem\demo\oemcore.mod\
xcopy io.112\io*.ssp        newchipset\oem\demo\oemio.mod\
xcopy obj\*.ssp             newchipset\oem\demo\oemchip.mod\
xcopy dim\pcipnp.ssp        newchipset\oem\demo\oemchip.mod\
xcopy gpc\power.ssp         newchipset\oem\demo\oemchip.mod\

cd newchipset     ←— 切到芯片层
xcopy sys\*.ssp             oem\demo\oemchip.mod
```

⑥ 修改/删减必要的源代码，并分离出有修改到的下列源代码于newchipset\sys目录：

```
MESSAGE.ASM  ←— 通过 EQUATOR.EXE 依照 ME9.EQU 所造出的.ASM 文件，供显示字
                符串信息控制之用。
EXTFUNC.ASM  ←— Setup 接口程序要调用的外部功能子程序
FLASH.ASM    ←— 针对该型芯片组的 Flash 读写控制子程序
POST.ASM     ←— 针对该型芯片组发展的开机自我检测 (POST)程序代码
MPPOST.ASM   ←— 针对多重处理器平台所需要的自我检测 (POST)程序代码
                (单处理器平台可不需要这段原始码)
MEMORY.ASM   ←— 针对该型芯片组发展的内存检测、芯片组寄存器 Initialize 的程序
RUNTIME.ASM  ←— BIOS Runtime 常驻服务函式模块，针对该型芯片组的修订版本
SETUP.ASM    ←—针对 SETUP 接口程序读写 CMOS 参数所需的接口钩结/拦截程序代码
```

其他未变动的*.ASM、*.EQU、*.INC等文件，则放到newchipset\SYS\NSYS的目录。

例如在本书所附光盘的\627.10\SIS530\SYS\目录中，最少要有这七个.ASM源代码文件，才能完成最基本的编译/链接作业。

⑦ 紧接着，从原先芯片层的 SYS 目录下，copy 一些源代码到特定的目录下面 ：

```
cd newchipset
copy sys\FLASH.ASM          oem\demo\oemport\OEMFLASH.ASM
copy sys\POST.ASM           oem\demo\oemport\OEMPOST.ASM
copy DIM\POSTDIM.ASM        oem\demo\oemport\OEMPDIM.ASM
copy sys\MEMORY.ASM         oem\demo\oemport\OEMMEMRY.ASM
copy sys\RUNTIME.ASM        oem\demo\oemport\OEMRUN.ASM
copy sys\SETUP.ASM          oem\demo\oemport\OEMSETUP.ASM
copy io\RMISC.ASM           oem\demo\oemio.mod\OEMRMISC.ASM
```

然后针对要修改的源代码以及拦截的跳跃点做修改，比如 cp_init → oem_cp_init，请参考第 5 章的 5-2 节 BIOS 执行流程图，然后自行修改。

⑧ 还要在 OEM 厂商层下的 OEMPORT 目录下，作几份必要的文件 ：

```
OEMRPCI.ASM  ←— 跟 RUNPCI.ASM 配合，做 PCI IRQ Routing 的程序代码 (稍后介绍)
OEMIOTBL.ASM ←— Super I/O 芯片的 I/O Initialization，IRQ 与 DMA 调配数据
```

有关于特定芯片组以及要修改的源代码实作部分，我们会在第 10 章硬件移植实战 DIY 中，再为各位读者做介绍。

快速移植/转换新的芯片组 BIOS 的方式

另一种比较直接的方式，就是以 Xcopy 的 DOS 指令，从先前已经实作成功的芯片层目录，将它下含整个子目录都复制成另一个新的目录。假使我要把既有的 SiS530 源代码目录转成 SiS540 芯片组，准备以既有的 SiS530 源代码来 porting 到 SiS540 芯片组，则可以执行下列指令复制整个目录，并以新的芯片组目录代号命名：

```
cd\627.10
xcopy sis530\*.* sis540\*.* /s/e
```

然后再根据需要来作目录以及源代码部分的删减、修改，这也是一项 BIOS 研发/维护人员可采用的程序之一。

例如我们想针对技嘉 GA-5SSM (也是 SiS530 芯片组结构)来 porting AMIBIOS，要如何为它作个新的 OEM 厂商层目录呢？

我们可以从将目前 OEM\DEMO 的整个 BIOS 源代码目录，整个转移到 DEMO3 这个目录：

```
xcopy sis530\oem\demo\*.* sis530\oem\demo3\*.* /s/e
```

然后以 GA5SSM 作为这个厂商/产品的计划目录就行了。在所附的光盘中，如 DEMO1， DEMO2 等目录，都是以这种方式移转，然后再仔细修改过的。

当然您也可以借助笔者所写的makeoem.bat来完成上述这些动作，它位于\627.10\sis530\oem\目录下，后面可以接你预计要porting的厂商 / 产品目录代码。例如：想要在\627.10\SIS530\OEM\目录下再作个称为 MYBOARD，只要执行下面的批处理命令即可：

```
C:\627.10\SIS530\OEM\>Makeoem myboard
```

这个批处理文件会自动建立myboard目录，并且把底下移植用的次目录，以及必要的源代码文件由核心层与芯片层去转移，这样可以简化读者Porting时，要依照627.10目录规则去作层层目录的麻烦。

快迅存取存储器芯片(Flash Part)的搭配

一般说来，主板厂商会依照它与零组件供货商往来的熟悉程度，参考当时采购的价格成本因素，来决定这一型主板产品，或者这一批产品要用哪些零组件规格。而快速存取芯片(Flash Part)，是一项变动性很高的零组件，可能一样都是生产Sis530主板，前面一万片都是用ATMEL 29C020的芯片，后两万片就有可能改用华邦(Winbond)29C020芯片来搭配出货。

还好在快速存取之间，许多型号与规格上都有相似甚至兼容、共通之处，差异点只在于烧写入电压、厂商与产品型号ID的不同，而BIOS厂商也极力提供完整的Flash芯片的读写子程序的支持，一般说来主板BIOS只要嵌入最常用到的几组Flash Part擦写程序，就足以应付各种不同Flash Memory芯片来安插，因此在这方面的问题并不大。

如果有比较新型的 Flash Memory 芯片，是目前 BIOS 程序代码不认得的，那么则是先针对该型 Flash 芯片写出两套程序代码，一套是给 Bootblock 激活区程序代码做自我应急 Flash 之用，另一套则是使用 ESCD/NVRAM，搭配 DMI 以及 PnP 作业时所使用。

将上述两套源代码编译为目标码 (*.asm→*.obj)之后，接下来要修改核心层 BIOS.MAK 以及 OBJ.MAK 维护文件，前者先添加这个新型 Flash 芯片的字段标志，后者则是修改程序叙述，把这两套程序代码以编译条件自动化的方式加入编译、连结，日后就可以用 MAKEALL.BAT，MAKEBIOS.BAT 来编译。

接下来照上述的方式，从 newchipset\sys 目录下的 FLASH.ASM 复制到 OEM 厂商层下的 OEMPORT 目录，并改名为 OEMFLASH.ASM。然后就下列子程序调用 / 进入点去修改。

OEMFLASH.ASM 要修改的程序进入点参考资料

子程序调用 / 进入点名称	用　途
MAKE_VPP_HIGH	Used to raise VPP (升高工作电压到可擦除 / 写入状态)
MAKE_VPP_LOW	Used to lower VPP (降低工作电压，此时成为只读模式)
OEM_FLASH_WRITE_DISABLE	将 Flash 芯片设为写保护模式的 OEM 特定的程序代码
OEM_FLASH_WRITE_ENABLE	将 Flash 芯片设为可擦除 / 写入模式的 OEM 特定的程序代码

最后一步则是将该型芯片的烧写规格与特性，加入 FLSINFO.ASM 以及 DFLSINFO.ASM，这样编译之后的 BIOS 监控程序，才能够跟这块新型号的 Flash 芯片做读写上的配合。

6-3 内存容量检测 / 计算

Memory Sizing(内存容量检测 / 计算)，是 BIOS 做移植以及除错作业时一项重要的关键。这部分的程序代码主要是由位于核心层 SYS 目录的 (\627.10\SIS530\SYS\MEMORY.ASM)，以及 OEM 厂商层下 OEMPORT 目录(\627.10\SIS530\OEM\DEMO\OEMPORT\OEMMEMRY.ASM)来决定。

在做 Memory Sizing 之前，我们要先对该型芯片组一些内存控制的规格、相关的 I/O 寄存器以及参数先有个了解。以本书我们针对 SIS530 芯片组的移植作业而论，这部分读者可以参考 6-1 节 的相关技术资料，或者到硅统的网站上取得关于 SIS530 芯片组的完整技术资料。

Memory Bank 设计数、实际数以及 DRAM 颗粒类型检测

相信读者如果组装过计算机的话，应该知道主板通常会设计二到四条 DIMM 的内存插槽，而一个内存插槽，最少会有 CS (Chip Select)，RAS (Row Address Strobe 列地址触发)，CAS (Column Address Strobe 行地址触发) 等信号；若是支持双面内存颗粒，则会有两组 CS 信号，而这样一个内存插槽的管理单位，我们习惯以一组 Bank(记忆库)来称呼。

而照业界的设计惯例与称呼，都是以一组能同时支持单面、双面颗粒的插槽管理单位称为一组 Bank，也就是将一组内存插槽拉入两组 CS 信号的线路设计，这样可以同时支持单面或双面内存颗粒的内存，而 BIOS 也可以检测得到。而芯片组通常都设计可以支持单、双面颗粒，不过是不是每一组 Bank 都具备双面的驱动能力，有时候要看该芯片组的设计特性，还有厂商要不要这么设计。这部分就是我们在 BIOS 做 Memory Sizing 时，要先掌握的关键。

如主板厂商因为设计上的需要，可以用其中一组双面 Bank 转拉出两组只支持单面颗粒的 DIMM 插槽，做出比较多组的 DIMM 插槽，这也就是读者会发现有些主板的某些内存插槽时，厂商声明只能支持单面内存模块的原因；如果硬插上双面颗粒，容量只能抓到一半，如图 6-1 所示。

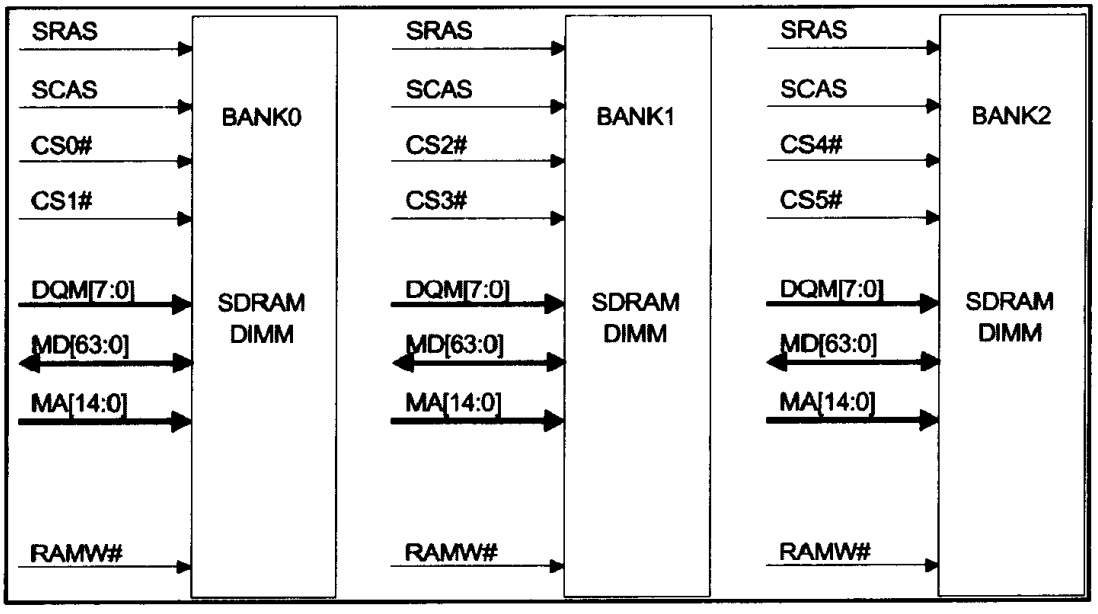


图 6-1　SIS530 内存控制器采用三组 Bank、双面的信号驱动结构

以 SIS 530 芯片组特性而言，可以设计三组支持单面、双面颗粒的 SDRAM Bank。DIMM0 在 PCI 组态寄存器索引地址是 060h，DIMM1 在 PCI 组态寄存器的索引地址是 061h，DIMM2 在 PCI 组态寄存器索引地址是 062h，而索引地址 63h 则是有关读取设置 DRAM 状态寄存器。

像 SIS530 这种支持三组、双面 Bank 的芯片组，厂商也可能把它设计成四只 DIMM 插槽，但是其中两只支持单、双面，两只只支持单面的内存插槽，甚至是六组只支持单面的内存插槽。不过这样设计都太占主板的面积，实际使用上也是限制重重，所以一般没有人会这样设计。最常见的就是厂商因为考虑到降低成本的因素，只设计两只 DIMM 插槽(规格没设计满)。

决定好了目前要设计、移植系统的 Bank 数之后，接下来就是检测实际上有几组内存模块插在 DIMM 插槽上，而且是哪几组 DIMM 插槽上有 DRAM？而 DRAM 颗粒的型态、时序等参数，一般是通过 SPD 数据以及系统芯片的检测，把状态值保存并且加以判断得来。

以下取自 \627.10\SIS530\SYS\POST.ASM 程序代码部分片段(DMI_MEM_MODULE_INFO)中，关于计算目前主板上有几组 Bank、使用的是单面、双面以及 DRAM 的型态：

```
;-------------------------------------------------------------------------------;
;              DMI_MEM_MODULE_INFO                                              ;
;-------------------------------------------------------------------------------;
; this routine is called from DMI POST INIT code to return information          ;
; about the concerned MEMORY MODULE.                                            ;
; input:                                                                        ;
;   AL   memory module# (1 based)                                               ;
;   stack available                                                             ;
; output:                                                                       ;
;    CY    routine not implemented                                            ;
;    NC    routine implemented                                                ;
;          AL    memory bank connection                                       ;
;                bit-7..4 = lower RAS line                                    ;
;                bit-3..0 = upper RAS line                                    ;
;                example: if bank connected to RAS 2&3, return (al)=23h       ;
;          AH    memory module current speed in ns                            ;
;                example: for 60ns memory return (ah)=3ch(=十进制60)          ;
```

起头以 1 为基准，第一组 DIMM 为 1，第二组 DIMM 为 2，以此类推

```
;           CX    current memory type (bit mapped)                              ;
;                 bit-15..11 = reserved (must be 0)                             ;
;                 bit-10= 1 -> SDRAM (同步内存 SDRAM 颗粒)                      ;
;                 bit-9 = 0                                                     ;
;                 bit-8 = 1 -> DIMM          (168pin, 64位的内存插槽)           ;
;                 bit-7 = 1 -> SIMM          (72pin, 32位的内存插槽)            ;
;                 bit-6 = 1 -> ECC           (具备 ECC 功能的DRAM)              ;
;                 bit-5 = 1 -> Parity        (具备 Parity功能的 DRAM)           ;
;                 bit-4 = 1 -> EDO           (EDO 内存)                         ;
;                 bit-3 = 1 -> fast page mode (FPM 内存)                        ;
;                 bit-2 = 1 -> standard                                         ;
;                 bit-1 = 1 -> unknown                                          ;
;                 bit-0 = 1 -> reserved (must be 0)                             ;
;           BL    installed memory size                                         ;
;                 bit-7 = 1 ->双面颗粒，0 -> 单面颗粒                           ;
;                 bit-6..0 = n 内存模块容量为2^n MB                             ;
;                 example: for 128MB memory module, bit-6..0 = 7                ;
;                 bit-6..0= 7Dh -> size unknown                                 ;
;                           = 7Eh -> reserved (do not use)                      ;
;                           = 7Fh -> not installed                              ;
; register usage : do not destroy any register except AX, BL, CX               ;
;-------------------------------------------------------------------------------;
    public dmi_mem_module_info
    public get_mem_module_info
get_mem_module_info:
dmi_mem_module_info:
    push  dx


;; set no RAS connect
    mov   bl, al          ; 取得输入的 DIMM 插槽数(开头以 1 起算)并放入BL
    dec   al              ; 转成对映的 Bank 数(开头以 0 为底)
```

```
    jnz     not_bank_0
    mov     ax, 0001h           ; 默认 Bank 0(第一组 DIMM 插槽)有连接
    jmp     @f
not_bank_0:
    test    al, 1
    jz      not_bank_1
    mov     ax, 0023h           ; 默认 Bank 0(第二组 DIMM 插槽)有连接
    jmp     @f
    not_bank_1:
    mov     ax, 00ffh           ; 默认 Bank 0(第三组 DIMM 插槽)有连接
@@:
    push    ax
    mov     al, bl              ; restore memory module #
    mov     cl, al              ; CL=memory module 号码(以 1 起算)
    mov     bl, al              ; 将以 1 起算的 memory module号码存到BL
    dec     bl
    mov     dl, bl              ; DL=BL=memory module#号码-1(以零为底)
;; for some transfer
    cmp     cl, 3
    jnz   not_dimm_3
    inc   cl
not_dimm_3:
;; detect dimm is install memory
    mov     ah, 63h             ; 从 index 63h(DRAM Status Register)
    call    read_pci_byte       ; 读取 DRAM DIMM 状态
    test    al, cl
    jz      dimm_empty
    mov     cx, 0500h           ; SDRAM, 168pin DIMM(bit10& bit8=1)
;; detect memory size in bank
    mov     al, bl
    call    get_bank_size       ; 调用 RUNTIME.ASM 中的 get_bank_size 子程序
    mov     bl, -1
```

```
not_zero_yet:
inc     bl
shr     ax, 1
cmp     ax, 0
jnz     short not_zero_yet

;; detect single side memory or double side(判断单面或双面颗粒模块)
add     dl, 60h
mov     ah, dl
call    read_pci_byte
and     al, 20h
shl     al, 2
or      bl, al

jmp   short check_memory_module_done
dimm_empty:
mov   ax, 000fh       ; speed unknown, no RAS connect
mov   bl, 7fh         ; memory not install
mov   cx, 0002h       ; memory unknown

check_memory_module_done:

pop   ax
pop   dx

clc
ret
```

以下是取自\627.10\SIS530\SYS\RUNTIME.ASM 一部分程序片段(也就是 get_bank_size 这个子程序) 的叙述 。

```
;;=============================================================
;;Input:al = 0, 1, 2
;;outut:ax = ? M size
;;=============================================================
        extrn   read_pci_byte:near
        public  get_bank_size
get_bank_size:
        push    bx
        push    cx
        push    dx
        push    si
        xor     dx, dx

        movzx   cx, al                  ;将 8 位 AL 数值转成 32 位数值存入 CX
        mov     ah, 63h                 ;从index 63h(DRAM Status Register)
        call    read_pci_byte           ;读取 DRAM DIMM 状态
        bt      ax, cx
        jnc     no_dram_installed

        mov     ah, 60h
        add     ah, cl                  ;读 index 60h+AL(0~2) 的 Bank 状态
        call    read_pci_byte
        mov     cl, al
        mov     ch, al                  ; CHCH for check DRAM side
        and     al, 0fh                 ; it is DRAM Type

;;the bank is SDRAM
        mov     bx, offset cgroup:sdram_size_translate_table
                                        ; 设置 SDRAM 转换表格偏移地址
get_size:
        db      2eh                     ; CS:
```

```
        xlat

        movzx dx, al

        shl     dx, 1

        test    ch, 00100000b
        jz      no_dram_installed       ;jmp if single size
        shl     dx, 1
no_dram_installed:
        mov     ax, dx                  ; output ax

        pop     si
        pop     dx
        pop     cx
        pop     bx
        ret

sdram_size_translate_table  label  byte
        db      4               ;no dram
        db      16              ;9*9
        db      16              ;10*10
        db      32              ;11*11
        db      08              ;12*12
        db      32              ;12*8
        db      32              ;12*9
        db      64              ;12*10
        db      16              ;12*11
        db      64              ;10*9
        db      64              ;11*9
        db      128             ;11*10
        db      8               ;11*10
sdram_size_translate_table_end label  byte
```

最后还有一项关于 DRAM Drivin Current (驱动电流值) 参数部分，一般说来芯片组都会有两组、三组甚至更多组电流值可以设置。如三条 DIMM 插槽只插一只 DRAM 时，所设置的驱动电流值可能要小一点；插满三只时，所送出的驱动电流值就要比较大一点，甚至插了某些“比较不稳定”的 SDRAM 内存模块时，电压值也得相对应做些调整。

这个部分由于牵扯到主板厂商产品的内部因素，如所设计的 PCB 电路板导电特性、供电零组件等，甚至即使是同一款芯片组，可能会因出厂批号不同，芯片厂会建议不同的电压 / 电流驱动值，这部分 BIOS 厂商以及笔者，无法帮您决定，您必须自行尝试、调配出一个最稳定的出厂默认值，然后写到 BIOS 成为默认值。

测试内存的方式

这个部分比较简单，一般的做法是直接对内存写入下列测试样板字组 (patterns)——如 5A5A5A5Ah，A5A5A5A5h 这几组 32 位字组，都是很适合作为测试内存容量的方式。第 3 章介绍过的 32bit (4GB) Flat Mode 模式，把 DS、ES 的段地址极限 (Segment Limit) 加大到 4GB，这样切回真实模式下，就可以在真实模式直接用 32 位寻址来读写、测试内存了。

以下是一个笔者所写的简单范例。

```
    mov    edi, 00200000h      ; 从 32MB 最小内存容量计算。读者也可以直
                               ; 接用 400000h，假定主板所支持的单只 DRAM
                               ; 容量为 64MB
test_a_region:
    mov es:[edi], 5a5a5a5ah    ; 确定 DS，ES 段地址极限已经调到 4GB (所谓
                               ; 32bit FLAT Mode)
    cmp es:[edi], 5a5a5a5ah
```

```
jne @
add edi, 00200000h          ;往上增加32MB (200000h)
cmp edi, 0FFE00000h         ;是否逼近4GB寻址空间的上限?
jb test_a_region            ;还没到,继续测试
or edi, 001FFFFFh           ;此时EDI数值即为检测到的DRAM地址边界上限
```

当然，这只是一个笔者随意写的小程序范例，真正在应用上，还要考虑到芯片组有没有支持 SMA (Share Memory Architecture，指绘图芯片的影像内存跟主板共享的结构)，此时你还得先做 VGA 芯片的 Frame Buffer Memory 的 sizing 动作，然后在这边加以扣除掉，才是目前系统 (主板)真实可以使用的内存容量。

On Board Cache 的形式侦测 / 容量检测方式

① 设定 Direct Access (或 Direct Map) 模式，如 Write Back 或 Write Through 模式。

② 检测 Cache 的形式，并且按照该形式设置必要的时序参数。

例如 asyschronous SRAM 时序为 X-3-3-3；Burst SRAM 时序为 X-2-2-2；Pipelined SRAM 则时序为 X-1-1-1 。

③ 计算 On Board Cache 内存的容量。

若芯片组有特殊寄存器机制，或者 Tag bits 可以读取出来，则 BIOS 程序代码直接通过芯片组来读取，或运用 tag bit 数来计算；若芯片组不支持 Cache 容量侦测，就只能用以依该芯片组所支持的最大 Cache 容量 (例如 2MB) 起算，逐步往下侦测出实际在主板上实际的 Cache 容量。

本书所介绍、示范的是 SiS530 芯片组结构的主板与 BIOS 系统，该平台具有 OnBoard Cache 的设计，自 Socket 7 结构之后的系统平台规划上，由于 CPU 都内建 L2 Cache，芯片组已经舍弃 On Board SRAM Cache的功能。

像 Intel Pentium Pro、Pentium II、Pentium III 与 Celeron 处理器所开发的 P6 Bus 芯片组，以及 AMD Athlon/Duron 等针对 Ev6 Bus 开发的芯片组，到截稿为止都还没有哪一家要做 On Board Cache 的设计；传闻 Micron 会针对 Athlon 推出内嵌 8MB eCache 的 Mamba 芯片组，但尚未推出。

所以读者可能接触到的 BIOS 源代码，特别是目前接触到 BIOS 维护的工程师，也许并没有这一段处理 Cache Sizing 的程序代码，或者在写法上可能又跟本书的范例有所不同。

读者如果有必要深入了解，可进一步参阅所附光盘 \627.10\SIS530\SYS\POST.ASM 这个源文件中，关于 GET_L2_CACHE_INFO 的子程序片段，以及 MEMORY.ASM 的程序代码最尾端 cache_test 处，也有一段关于 Cache sizing 的程序代码范例可供参考。

Memory/Cache Sizing 的源代码浏览

接下来，让我们实际看一下由 AMI 公司所提供的 AMI BIOS for SIS530 chipset 中，对于 Memory Bank、DRAM type 的检测，以及 Cache test & sizing 的程序代码浏览，笔者已将部分原文批注翻译成中文，有些未加批注之处笔者也尽可能加入解释。希望能有助于读者以及 BIOS 维护工程师，对这部分的流程能够大致了解并快速掌握。

文件名	MEMORY.ASM
路径	\627.10\SIS530\SYS\MEMEORY.ASM 或..\..\SYS\MEMORY.ASM (相对于 OEM 厂商层)

```
page  ,132
title  CHIPSET INIT/MEMORY SIZING
;-------------------------------------------------------------------------------;
; NOTE: Do not destroy EBP, FS, GS, SS, DS, ES unless otherwise specified.  ;
;-------------------------------------------------------------------------------;
;*******************************************************************************;
..... 版权声明。此部分叙述省略 .....;
*******************************************************************************;
;---------------------------;
      include mbiosmac.mac ;
      include mbiosequ.equ  ;   >引入先前定义的宏变量定义文件
      include makeflag.equ  ;
;---------------------------;
      extrn   u_sis5595_pci2isa_init_table:byte     ;
      extrn   u_sis530_init_table:byte              ;

      extrn   read_sio_byte_x:near                  ;
      extrn   read_pci_byte_x:near                  ;   声明需要调用其他目标码(*.obj)
      extrn   write_sio_byte_x:near                 ;   已公开的子程序进入点
      extrn   write_pci_byte_x:near                 ;
      extrn   cmos_data_in:near                     ;
      extrn   cmos_data_out:near                    ;
;---------------------------------------;
cgroup        group  _text
_text segment word  public 'CODE'
assume        cs:cgroup
.586p
;---------------------------------------;
      public _MEMORY_STARTS
_MEMORY_STARTS label  byte ; 标示 MEMORY 这个模块的开始地址
```

```
;-------------------------------------------------------------------------------;
; U_POWER_ON_INIT (BIOS 开机初始进入点)                                          ;
;-------------------------------------------------------------------------------;
; 此进入点会在计算机一开机由 BIOS 刚获得控制权之后调用，除了 EDX 为              ;
; PUID 数值之外，进入这个进入点之前不需要定义任何参数。详细资料请看;
; 源文件的说明                                                                  ;
; 注意事项   1. 进入这个进入点时 Stack 还不能使用，必须用特殊的调用            ;
;               方式处理。                                                      ;
;            2. 这个进入点的程序片段负责可以打开 RTC/KBC，如果芯片             ;
;               组已经内建 RTC/KBC 功能；若厂商决定采用外部 Super I/O ;
;               芯片的 RTC/KBC，请不要在这个程序代码片段做任何                 ;
;               initializing 的动作。                                           ;
;-------------------------------------------------------------------------------;
    public  u_power_on_init          ;声明本程序代码 u_power_on_init 子程序为公用性质
    public  oem_u_power_on_init_end  ;宣告 oem_u_power_on_init_end 子程序为公
                                     ;开性质
    extrn   oem_u_power_on_init:near ;引用外部 oem_u_power_on_init
                                     ;(在 OEMMEMRY.ASM)
    extrn   u_power_on_init_end:near ;引用外部 u_power_on_init_end
                                     ;(在 OEMMEMRY.ASM)
u_power_on_init:
    mov     ebp, edx                 ;Save CPUID in EBP (将CPU值存到 EBP)
    mov     si, offset cgroup:u_sis5595_pci2isa_init_table
                                     ;SI 指向 sis5595 南桥芯片初始参数表偏移地址
u_sio_init:
    mov     ah, cs:[si]
    or      ah, ah
    jz      u_sio_init_done          ; 读到 00 表示读到最后一个参数，跳到
                                     ; u_sio_init_done 初始完毕程序处继续执行

jmp_di read_sio_byte_x               ; 利用 Jmp_di 模拟 Call read_sio_byte_x
                                     (从南桥芯片读入一个 byte 到 al)
    inc     si                       ;SI=SI+1
    and     al, cs:[si]              ; 跟第二组 AND 操作数先做 AND 处理
    inc     si                       ; SI=SI+1
    or      al, cs:[si]              ; 跟第二组 OR 操作数先做 OR 处理
```

```
    inc   si
    jmp_di write_sio_byte_x      ;利用jmp_di宏仿真call write_sio_byte_x
                                 ;(将处理过的AL参数写到南桥芯片)
    jmp   short u_sio_init       ;继续读入/处理下一处参数值直到读入00为止
u_sio_init_done:

    mov   ah, 45h                ;Open APC
    jmp_di read_sio_byte_x
    mov   bx, ax                 ;Save Reg.45h to BX
    or    al, 02h
    jmp_di write_sio_byte_x

    mov   ax, 8787h              ;APC Register 07h
    ret_sp cmos_data_in
    and   al, 0feh               ;Clear KPR_LST
    xchg  al, ah
    ret_sp cmos_data_out         ;以ret_sp宏仿真call cmos_data_out子程序
    mov   ax, 8383h              ;APC Register 03h
    ret_sp cmos_data_in
    or    al, 040h               ;Enable APC Function
    and   al, not 2ch            ;Disable Ring/RTC/STREQ0
    xchg  al, ah
    ret_sp cmos_data_out         ;以ret_sp宏仿真call cmos_data_out子程序
    mov   ax, bx                 ;Close APC
    jmp_di write_sio_byte_x      ;Restore Reg.45h

    mov   si, offset cgroup:u_sis530_init_table
                                 ;SI指向sis530北桥芯片初始参数表偏移地址
    u_pci_init:
    mov   ah, cs:[si]
    or    ah, ah                 ;读到00表示读到最后一个参数，跳到
    jz    u_pci_inti_done        ;u_pci_init_done初始完毕程序处继续执行
    jmp_di read_pci_byte_x
    inc   si                     ;SI=SI+1
    and   al, cs:[si]            ;跟第二组AND操作数先做AND处理
```

```
    inc     si                      ; SI=SI+1
    or      al, cs:[si]             ; 跟第三组 OR 操作数做 OR 处理
    inc     si                      ; SI=SI+1
    jmp_di write_pci_byte_x         ; 以 jmp_di 宏仿真 call write_pci_byte_x, 将
                                    ; AL 参数值写入 530 北桥芯片, 并返回下一行
    jmp     short u_pci_init        ; 继续读入下一组参数直到读入 00 为止
u_pci_inti_done:

; set all IRQs to edge trigger (设置所有 IRQ 信号中断之产生方式为边缘触发)
    xor     ax, ax
    mov     dx, 04D0h
    out     dx, al
    jmp     short $+2
    inc     dx
    out     dx, al
    jmp     short $+2

; Do OEM specific stuff
    jmp oem_u_power_on_init ; 跳入OEMMEMRY.ASM 中的 oem_u_power_on_init 处
oem_u_power_on_init_end:  ; 从OEMMEMRY.ASM 程序片段执行结束会回到这里
    mov     edx, ebp            ; 将原先 CPUID 值存回 EDX 寄存器
    jmp     u_power_on_init_end  ; 跳回UB.ASM中的u_power_on_init_end处
;-------------------------------------------------------------------------------;
; U_CP_INIT (BIOS 开始做芯片组启始化的程序片段)                                  ;
;-------------------------------------------------------------------------------;
; 侦错检查码:  D3                                                               ;
; 输入参数:       DS=ES=0000h, set to 4GB limit(Flat Mode)                      ;
;                 SS=CS,因此时 DRAM refresh 功能尚未激活,Stack 堆栈             ;
                  还无法使用                                                    ;
; 注意:           若 DRAM refresh 功能由 DMA Timer 控制, 并非由芯片             ;
                  组另外控制, 则此时可通过 DMA Timer 激活 refresh               ;
                  功能。                                                        ;
;-------------------------------------------------------------------------------;
    public  u_cp_init
    public  oem_u_cp_init_end
```

```
        extrn   u_cp_init_end:near
        extrn   oem_u_cp_init:near
u_cp_init:
        jmp     oem_u_cp_init           ;跳至OEMMEMRY.ASM中的oem_u_cp_init
oem_u_cp_init_end:                      ;从OEMMEMRY.ASM返回到这一行
        if(MKF_NCPU eq 2)               ;
        jmp     u_init_local_apic       ;若本主板支持双CPU，则跳至 u_init_local_apic
u_init_local_apic_end:                  ;针对 local APIC 芯片进行 initializing 的动作
        endif                           ;
        jmp     cas_latency_set
cas_latency_set_end:

        jmp     cp_init_x
cp_init_x_end:

        jmp     cache_test              ;return bl
cache_test_end:

%out=Memory refresh starts after memory sizing=   ;目前编译到这里，送出一段
                                                  ;信息以便日后除错之用
;-----------------------------------------------------------------------;
; START MEMORY REFRESH (激活内存周期性刷新功能)  ;
;-----------------------------------------------------------------------;
        extrn   refresh_low_value:byte
        mov     al, 00h                 ;起始 DMA 页寄存器(DMA PAGE register)
        out     8fh, al                 ;(port 8Fh)
        mov     al, 01010100b           ;指定第一组计时 channel(系统保留为 DRAM refresh)
        out     timer_control_port, al  ;送到 8254 port 43h 计时控制端设置计时间隔
        io_delay
        mov     al, cgroup:refresh_low_value        ;取得默认的 refresh 计时数值低位
        out     timer_1_port, al                    ;8254 Port 41h(Channel 1)
        mov     cx, 100h
        xor     di, di                              ;设置 EDI=00000000h
        mov     es, di                              ;ES = DI = 0000h
wpulse1:
```

```
    stosb
    loop    wpulse1

    jmp     u_cp_init_end           ;跳到UB.ASM的u_cp_init_end
;---------------------------------------  ;
sdram                       = 1
;---------------------------------------  ;
DATA_PATTERN1 EQU   012345678H  ;第一组测试pattern
DATA_PATTERN2 EQU   087654321H  ;第二组测试 pattern
DATA_PATTERN3 EQU   05a5a5a5aH  ;第三组测试 pattern
DATA_PATTERN4 EQU   0a5a5a5a5H  ;第四组测试 pattern
;-----------------------------------------;
;*************************************************************************;
PushNS      macro       Reg
            shl                 Reg, 16
            endm
;*************************************************************************;
PopNS       macro       Reg
            shr                 Reg, 16
            endm
;*************************************************************************;
;---------------------------------------;
; AW_DELAY MACRO DEFINITION   ;定义一个延迟的宏指令(aw_delay)
;---------------------------------------;
aw_delay macro
    out     0e0h, al
endm

init_dram label byte                    ; initilizing DRAM 的参数表标记地址
    db      056h, 002h

    db      063h, 007h

    db      05ch, 010h
    db      05ch, 090h
```

```
        db      05ch, 050h
        db      05ch, 000h
        db      05ch, 020h
        db      05ch, 020h
        db      05ch, 020h
        db      05ch, 020h
        db      05ch, 020h
        db      05ch, 020h

        db      063h, 004h
        db      0ffh

cp_init_x:
;;=================>init the initial reg value
mov si, offset cgroup:init_dram      ;从索引寄存器 60～6Fh 都送出初始参数，准备
                                     ;做 memory sizing
id_loop_1:
        mov     ah, byte ptr cs:[si]
        cmp     ah, 0ffh
        jz short id_loop_1_end
        mov     al, byte ptr cs:[si+1]
        jmp_di write_pci_byte_x
        inc     si
        inc     si
        jmp     short id_loop_1
id_loop_1_end:

;=============================================================================;
;       EXTRN           csMem_MatrixJBX:near
;       EXTRN           csMem_MatrixsJBX:near
;       EXTRN           csPattern_TestJBX:near
;       EDO_DRAM        =       1
;       SDRAM           =       1
;       做 memory size 之前，先默认 bank 0, 1, 2 为 Fast Page Mode 内存颗粒、
;        type3、单面且内存总线宽度为 64 位，而 DS、ES 段基地址为 0，
;       段极限都调到 4G(32big Mode)
```

```
STARTDRAMSIZE:
    MOV   AH, 062H              ;bank pointer
;-------------------------------------------------------------;
BANKSTART:
    mov   dx, ax
    pushns edx                  ;save bank pointer(把 bank 指位器做个备份到edx)
;-------------------------------------------------------------;
jmp_di read_pci_byte_x
    or    al, 00bh              ;low nibble already '3'
    jmp_di write_pci_byte_x     ;Pre-set SDRAM present(默认 DRAM 形式为SDRAM)
;-------------------------------------------------------------;
ram_test_begin:
    mov   al, 0                 ;默认 DRAM 的 ma_type 值为 0
    jmp_bx cspattern_testjbx    ;以 jmp_bx 宏仿真 call cspattern_testjbx
                                ;并返回下一行(要先确定最少有一组 DRAM 模块存在)
    jnz   bankdone              ;and put data pattern into offset 0
jmp_bx csmem_matrixsjbx         ;output=al (actully ma_type+1)
BANKDONE:
;-------------------------------------------------------------;
    dec   al

    popns edx                   ;延迟一段时间后将 dram type 值取回edx
    mov   dl, al
    pushns edx

    cmp   al, 0ffh              ;若 al=0ffh 表示抓不到 dram type
    jz    no_dram               ;

    mov   al, 2bh
    jmp_di write_pci_byte_x

normal_dram_dside_detect:
    xor   edi, edi              ;测试另一边(以 256MB 为边界)
    mov   dword ptr es:[edi+10000000h], Data_pattern1      ;write 256M+0
    mov   dword ptr es:[edi+10000008h], Data_pattern2      ;write 256M+8
```

```
        cmp     dword ptr es:[edi+10000000h], Data_pattern1
        jz      double_side_prsnt

clear_dside_dram:
        and     al, not 020h
        jmp_di write_pci_byte_x

double_side_prsnt:
no_dram:
        popns   edx
        mov     ax, dx
        push    edx

        mov     bl, al
        jmp_di read_pci_byte_x          ; 读入目前指定 Bank 的相关状态信息
        cmp     bl, 0ffh
        jz      should_clear_cfg
        and     al, 0f0h
        or      al, bl
        jmp     @f
should_clear_cfg:
        mov     cl, ah                  ; 取得正确的 bank enable bit
        and     cl, 07h                 ; 忽略 bit7
        mov     bh, 1
        shl     bh, cl
        mov     bl, ah

        mov     ah, 63h
        jmp_di read_pci_byte_x          ; 设置 Sis530 DRAM status reg 并读入状态值
        not     bh
        and     al, bh                  ; 清除 Bank 准备测试
        jmp_di write_pci_byte_x

        mov     ah, bl
        mov     al, 0
```

```
@@:
    jmp_di write_pci_byte_x          ;将这个 Bank 的设置值写入
    dec    AH            ;DECREASE BANK POINTER(准备处理下一个 Bank)
    cmp    ah, 60h       ;比较是否处理到 bank0(register index 60h)
    jb     sizing_done

    mov    dx, ax
    pushns edx

    mov    cl, ah                ;取得正确的 bank enable bit
    and    cl, 07h
    mov    bh, 1
    shl    bh, cl

    mov    ah, 63h               ;指定 DRAM Status register(index 63h)
    jmp_di read_pci_byte_x
    or     al, bh
    jmp_di write_pci_byte_x      ;将 bank enable bit 写到 DRAM status register
bank_finish:
    popns  edx
    mov    ax, dx

    jmp    bankstart

sizing_done:
    mov    ah, 063h
    jmp_di read_pci_byte_x

    test   al, 07h
    Jnz    DRAM_SIZING_DONE
    xor    ecx, ecx
@@: in     al, 80h               ;NO DRAM COMES HERE
    inc    al
    out    80h, al
    db     67h
```

```
    loop    $
    jmp     @b
DRAM_SIZING_DONE:
;;=============================================================
    jmp  cp_init_x_end                  ;end

;---------------------------------------  ;
set_ma_type macro       x
            xor         al, al
            or          al, x
            endm
;---------------------------------------  ;
;output ZF=0 means no dram (ZF=0 代表没有任何 DRAM 安插)
;     for 64/32 mix system(针对 64、32 位内存总线混合结构的主板)
;           CF=0 ◄── 代表内存总线为 64 位，=1 →代表内存总线为 32 位
;     for pure 64 system (单纯 64 位内存总线的平台)
;           CF=0 ◄── 代表内存颗粒测试正常，1 代表内存颗粒测试不正常
csPattern_TestJBX   proc  near  public
MOV         DWORD PTR ES:[00h], DATA_PATTERN1 ;将 Pattern1 写到内存地址
aw_delay
jcxz        $+2
MOV         DWORD PTR ES:[08h], DATA_PATTERN2       ; 把Big Endian的Pattern
                                                    ; 写入下个偏移64 位
                                                    ; 的内存地址，有清除
                                                    ; 总线(Clear Bus) 的
                                                    ; 作用
aw_delay
jcxz        $+2
CMP         DWORD PTR ES:[00h], DATA_PATTERN1       ;MEMORY EXIST(此处
                                                    ;Pattern 是否还存在)?
aw_delay
jcxz        $+2
jnz         @f

MOV         DWORD PTR ES:[04h], DATA_PATTERN3       ;WRITE MEMORY(将另
                                                    ;一个测试Pattern 写入)
```

```
    aw_delay
    jcxz  $+2
    MOV  DWORD PTR ES:[0ch], DATA_PATTERN4        ;CLEAR BUS(运用上述
                                                  ;的技巧清除总线)
    aw_delay
    jcxz   $+2
    CMP  DWORD PTR ES:[04h], DATA_PATTERN3        ;MEMORY EXIST(此处
                                                  ;Pattern 是否还存在)?
    aw_delay
    jcxz   $+2
    jnz    dram_32_present
    clc
@@:
    jmp    bx
dram_32_present:
    cmp    sp, sp
    stc
    jmp    bx
csPattern_TestJBX        endp

;---------------------------------------;
;---------------------------------------;
SMA_C8   equ   00000800h
SMA_C9   equ   00001000h
SMA_Cc   equ   00004000h

SMA_Rc   equ   00004000h
SMA_Rd   equ   04000000h
SMA_Re   equ   08000000h
;---------------------------------------- ;
sma_none          equ   0000b        ;
sma_1_11_08       equ   0001b        ;0
sma_1_13_08       equ   0010b        ;1
sma_1_11_09       equ   0101b        ;4
sma_1_13_09       equ   0110b        ;5
```

```
sma_1_11_10     equ     1001b       ;8
sma_1_13_10     equ     1010b       ;9
sma_2_11_08     equ     1101b       ;c
sma_2_12_08     equ     0011b       ;2
sma_2_13_08     equ     0100b       ;3
sma_2_12_09     equ     0111b       ;6
sma_2_13_09     equ     1000b       ;7
sma_2_12_10     equ     1011b       ;a
sma_2_13_10     equ     1100b       ;b
;-------------------------------------;

csMem_MatrixSJBX    proc  near  public
    xor     edi, edi
;-------------------------------------------------------------;
; asymmetry dram detecting should be put here(测试单、双面颗粒的程序叙述)
    CMP   DWORD PTR ES:[edi+sMA_Cc], DATA_PATTERN1
    jcxz  $+2                                ;test a10
    Jne   two_bank_sdram                     ;mac8 valid , jmp
    CMP   DWORD PTR ES:[edi+sMA_C8], DATA_PATTERN1
    jcxz  $+2
    jne   one_bank_10c

    set_ma_type  sma_1_13_08

    CMP   DWORD PTR ES:[edi+sMA_Rd], DATA_PATTERN1
    jcxz  $+2
    jne   @f
    set_ma_type  sma_1_11_08
@@:  jmp  bx
one_bank_10c:
    CMP   DWORD PTR ES:[edi+sMA_C9], DATA_PATTERN1
    jcxz  $+2
    jne   one_bank_11c

    set_ma_type  sma_1_13_09
```

```
    CMP   DWORD PTR ES:[edi+sMA_Rd], DATA_PATTERN1
    jcxz   $+2
    jne    @f
    set_ma_type   sma_1_11_09
@@:  jmp   bx
one_bank_11c:
    set_ma_type   sma_1_13_10
    CMP   DWORD PTR ES:[edi+sMA_Rd], DATA_PATTERN1
    jcxz   $+2
    jne    @f
    set_ma_type   sma_1_11_10
@@:  jmp   bx

    two_bank_sdram:
    CMP   DWORD PTR ES:[edi+sMA_C8], DATA_PATTERN1
    jcxz   $+2
    jne    two_bank_10c

    set_ma_type   sma_2_13_08
    CMP   DWORD PTR ES:[edi+sMA_Re], DATA_PATTERN1
    jcxz   $+2
    jne    @f
    set_ma_type   sma_2_12_08
    CMP   DWORD PTR ES:[edi+sMA_Rd], DATA_PATTERN1
    jcxz   $+2
    jne    @f
    set_ma_type   sma_2_11_08
@@:  jmp   bx
two_bank_10c:
    CMP   DWORD PTR ES:[edi+sMA_C9], DATA_PATTERN1
    jcxz   $+2
    jne    two_bank_11c

    set_ma_type   sma_2_13_09
    CMP   DWORD PTR ES:[edi+sMA_Re], DATA_PATTERN1
```

```
    jcxz    $+2
    jne     @f
    set_ma_type   sma_2_12_09
@@:  jmp   bx
two_bank_11c:
    set_ma_type   sma_2_13_10
    CMP   DWORD PTR ES:[edi+sMA_Re], DATA_PATTERN1
    jcxz    $+2
    jne     @f
    set_ma_type   sma_2_12_10
@@:  jmp   bx
    csMem_MatrixSJBX    endp

;*************************************************************************;
;*************************************************************************;
cas_latency_set:
    mov     ah, 45h             ; 指定使用扩展的 CMOS 内存
    jmp_di read_sio_byte_x
    or      al, 08h
    jmp_di write_sio_byte_x

    mov     al, 80h             ; read external cmos offset 0h, bit0
    ret_sp  cmos_data_in        ; 读取扩展 CMOS 的偏移地址 0, bit0
    db      0d4h, 10h           ; AAM 10h
    xor     ah, al
    cmp     ah, 0fh
    jnz     set_cas_latency_done
    test    al, 1
    jnz     set_cas_latency_done

    mov     ah, 05dh
    jmp_di read_pci_byte_x
    and     al, not 08h         ; 设置 CAS Latency 为 2
    jmp_di write_pci_byte_x
```

```
    set_cas_latency_done:
    mov   ah, 45h                    ; Close external cmos
    jmp_di read_sio_byte_x           ; 关掉扩展的 CMOS 内存
    and   al, not 08h
    jmp_di write_sio_byte_x

    jmp   cas_latency_set_end

;--------------------------------------;
burst_sram          = 0
;--------------------------------------;
;   CACHE SIZING (On Board Cache Sizing)
;--------------------------------------;
....本程序代码片段与作用批注省略 (有兴趣读者请参考源代码原文批注)....
.....
;-------------------------------------------------------------------------------------;
;         INIT_ATAPI_HARDWARE (初始化 ATAPI 装置的子程序)                              ;
;-------------------------------------------------------------------------------------  ;
....本程序代码片段与作用批注省略 (有兴趣读者请参考源代码原文批注)....
.....
;*************************************************************************************;
..... 版权声明。此部分叙述省略 .....
;*************************************************************************************;
;-------------------------------------------------     ;
if (MKF_NCPU eq 2)
    include uapic.inc
endif
;-------------------------------------------------     ;
    public _MEMORY_ENDS
    _MEMORY_ENDS  label  byte                ; MEMORY程序模块的结束标记
;-------------------------------------------------     ;
_text       ends
            end
```

OEMMEMRY.ASM

OEMMEMRY.ASM是AMI BIOS程序代码开放给OEM厂商自行修改、置换或添加新程序代码的源代码。从AMI技术文件中，OEMMEMRY.ASM与前面MEMORY.ASM都是侦错检查点0D3h会涵盖到的区域。

一般说来，OEM厂商多半只会动到这个OEMMEMRY.ASM，来针对新型的内存做Initializing、setting等动作，或者排除冲突、兼容性的issue。底下是OEMMEMRY.ASM要修改的程序进入点资料：

子程序调用进行点名称	用　途
OEM_U_CP_INIT	针对某芯片组特性所添加的初始化程序代码与内存检测码的修改点。从MEMORY.ASM源代码的U_CP_INIT调用。
OEM_U_POWER_ON_INIT	OEM厂商想在Power on之后要修改、加入新功能的程序代码修改点。此处从MEMORY.ASM源代码中U_POWER_INIT处调用。

最后，我们也将这个OEMMEMRY.ASM的源代码列举出来，供读者做浏览上的参考。

文件名	OEMMEMRY.ASM
路径	\627.10\SIS530\OEM\DEMO\OEMPORT\OEMMEMRY.ASM或 OEMPORT\OEMMEMRY.ASM (相对于OEM厂商层)

```
        page    ,132
        title   OEM HOOKS AT POWER-ON AND MEMORY SIZING
;--------------------------------------------------------------------;
; NOTE: Do not destroy EBP, FS, GS, SS, DS, ES unless otherwise specified.
;--------------------------------------------------------------------------------;
```

```
;*****************************************************************************;
;..... 版权声明。此部分叙述省略 .....
;*****************************************************************************;
;-------------------------------------------------------------  ;
        include mbiosmac.mac
        include makeflag.equ
;-------------------------------------------------------------  ;
        public          u_sis5595_pci2isa_init_table
        public          u_sis530_init_table
;-------------------------------------------------------------  ;
cgroup group _text
_text segment word public 'CODE'
        assume cs:cgroup
.486p
;-------------------------------------------------------------  ;
        public _OEMMEMRY_STARTS
_OEMMEMRY_STARTS    label byte          ;OEMMEMRY 模块的初始标记地址
;-------------------------------------------------------------------------------;
; OEM_U_POWER_ON_INIT(给OEM 附加的OEM_POWER_ON_INIT 程序进入;
;点 / 片段)                                                                     ;
; 输入参数：EDX=CPU ID 值, SS=CS, 此时 Stack 堆栈尚不能使用                     ;
; register usage : can destroy any register except EDX, SS, FS, GS              ;
;-------------------------------------------------------------------------------;
        public  oem_u_power_on_init
        extrn   cmos_data_in:near
        extrn   write_pmu_byte_x:near
        extrn   read_pmu_byte_x:near
        extrn   oem_u_power_on_init_end:near
        extrn   read_sio_byte_x:near
        extrn   write_sio_byte_x:near
oem_u_power_on_init:
        mov     ax, 07388h
        jmp_di write_pmu_byte_x

;; turn off ACPI led
        mov     ah, 045h
```

```
    jmp_di read_sio_byte_x
    or      al, 02h
    jmp_di write_sio_byte_x

    mov     al, 04h
    out     70h, al
    jmp     short $+2
    in      al, 71h
    jmp     short $+2
    or      al, 08h
    and     al, not 03h
    jmp     short $+2
    out     71h, al

    mov     ah, 045h
    jmp_di read_sio_byte_x
    and     al, not 02h
    jmp_di write_sio_byte_x

    jmp     oem_u_power_on_init_end
;-------------------------------------------------------------------------------;
;OEM_U_CP_INIT (开放给OEM附加的OEM_U_CP_INIT程序进入点/片段)                    ;
;-------------------------------------------------------------------------------;
;                                                                               ;
; 侦错检查点：D3                                                               ;
; 此子程序片段被 U_CP_INIT 处理完 memory sizing 之后调用，可以供               ;
; OEM修改/添加特殊 initialization 初始码做特定的应用                           ;
; 寄存器/参数要件：(详情请参考所附光盘的英文源代码)                            ;
; 1 DRAM 的 refresh 机制已激活                                                 ;
; 2 SS=CS，但 Stack 尚未设置好                                                 ;
; 3 DS 与 ES 均设为 0000h (DS=ES=0000h), DS 与 ES 段地址极限已调整为4GB        ;
;-------------------------------------------------------------------------------;
    public oem_u_cp_init        ;宣告这个 oem_u_cp_init 子程序为公开性质
    extrn   oem_u_cp_init_end:near
oem_u_cp_init:
    if(MKF_ONCHIP_LM78)
```

```
;; Start LM78 Function
        mov     al, 40h
;       mov     dx, 295h
        mov     dx,MKF_LM78_BASE_ADDRESS    ; 由IO.MAK 中的 LM78_BASE_
                                            ; ADDRESS 值造出 MKF_LM78_
                                            ; BASE_ADDRESS 这个常数
        out     dx, al
        jmp     short $+2
        mov     al, 1
        inc     dx
        out     dx, al
endif

        jmp     oem_u_cp_init_end ;跳回MEMORY.ASM中的 oem_u_cp_init_end 处
;-------------------------------------------------------------------------;
;       INITIALIZATION TABLES USED IN U_CP_INIT                           ;
;   Put all the tables which are used in U_CP_INIT hook in MEMORY.ASM     ;
;   and may be different for different OEMs.                              ;
;-------------------------------------------------------------------------;
u_sis5595_pci2isa_init_table label byte; sis5595 南桥 (PCI 对 ISA 桥接)
;芯片参数表参数数据格式 (第一组寄存器索引值，第二组为 AND 屏蔽运算
;值，第三组为 OR 屏蔽运算值)
;           Reg.   AND   OR

if MKF_256K_BIOS
    db      040h,  000h,  0cbh          ; 128KB BIOS 与 256KB BIOS 差别在于
else                                    ; index 40h 要送出的参数 bit0=1 (128KB)
    db      040h,  000h,  0cah          ; 或 bit0=0 (256KB)
endif
    db      045h,  000h,  020h
    db      048h,  008h,  0f7h
if (MKF_USB_SUPPORT)
    db      062h,  030h,  0c0h          ; usbirq enable
else
    db      062h,  030h,  080h          ; usbirq disable
endif
```

```
        db      064h,  000h,  034h
        db      06dh,  0fdh,  002h          ;Enable I2C
        db      070h,  0e3h,  01ch
        db      07bh,  0efh,  080h          ;Enabled LM78
        db      091h,  000h,  MKF_PM_BASE_HIGH;ACPI_BASE_ADDRESS位于高位
        db      090h,  03fh,  MKF_PM_BASE_LOW;ACPI_BASE_ADDRESS位于低位
        db      004h,  0fch,  003h
        db      000h                        ;最后补0，供初始参数的子程序辨识之用
u_sis5595_pci2isa_init_table_end label byte

u_sis530_init_table label byte              ;sis530北桥芯片初始参数表地址
;               Reg.   AND   OR
        db      051h,  0bfh,  00h           ;
;       db      055h,  030h,  0bh           ;
        db      055h,  030h,  0dh           ;
        db      058h,  0afh,  050h          ;
        db      087h,  011h,  044h          ;
        db      05dh,  0f7h,  008h          ;默认CAS Latency为3
        db      064h,  001h,  000h          ;Memory Buffer Strength
        db      064h,  0fch,  000h          ;(内存插槽电流强度)
        db      06bh,  000h,  MKF_PM_BASE_HIGH
        db      06ah,  01eh,  MKF_PM_BASE_LOW+1
        db      076h,  017h,  008h          ;!!Legacy USB!!
        db      004h,  0fdh,  002h          ;Enable Memory Space
        db      000h                        ;最后补0，供初始参数的子程序识别
u_sis530_init_table_end label byte

;*****************************************************************************;
;.....版权声明。此部分叙述省略.....
;*****************************************************************************;
;-----------------------------------------------------------;
        public  _OEMMEMRY_ENDS
_OEMMEMRY_ENDS label  byte                  ;OEMMEMRY.ASM的结束地址标记处
;-------------------------------------      ;
_text   ends
        end
```

6-4 PCI IRQ 号码配置与绕线设置

何谓 PCI IRQ Routing (PCI 中断号码配置与绕线设置)呢？我们要先从 PCI 装置的中断信号处理方式简单地讲起。

如果各位读者手边有相关 PCI 系统结构的书籍的话，应该知道在一个单纯的 PCI 总线系统中，若无其他 PCI 总线桥接控制芯片下，PCI 装置最多能有 32 个相互连接，这里所谓的装置 (device)可以是主板上的芯片(如南桥、北桥芯片、USB 芯片、SCSI、网卡芯片等等)，一个 PCI 扩充槽也是占用一个装置的扩展，当这个扩充槽插上 PCI 适配卡时，它就占用了一个 PCI Device 的名额。通常在主板中，芯片组本身最少就占一组(PCI 控制器)，有时候会占用到两组、三组甚至更多，如图 6-2 所示。

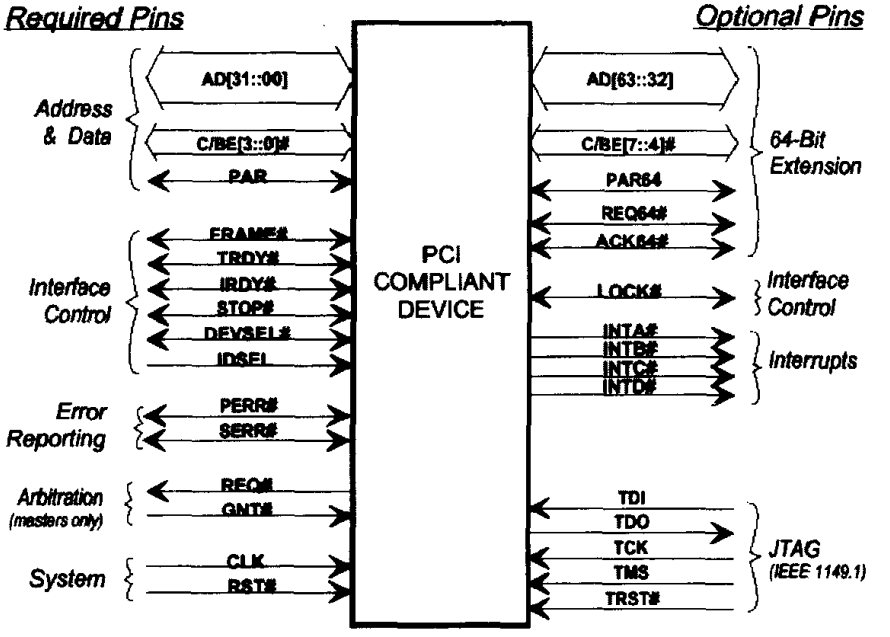


图 6-2 PCI 信号脚位示意图 (取自 PCI Local Bus Specification Revision 2.2 文件)

虽然PCI装置最多有32组，可能是为了信号上的定义、负荷或其他因素，一般说来最常见的PCI装置绕线设计方式，就是以AD11～AD31地址线连接到IDSEL信号选择线，也就是说以AD11来连接、触发第一组PCI装置的(IDSEL/DEVSEL)，AD12连接到第二组PCI装置的(IDSEL/DEVSEL)信号线，这种线路连接法则，在一个32位PCI信号群中，总共有AD11～AD31也就是最多有21组PCI装置(含北桥、南桥、PCI扩充槽等等)，可独立被选择、读写和组态。

PCI适配卡所使用的中断信号处理的方式，跟传统PC从AT时代的ISA结构有很大的不同。PCI装置对于中断信号的定义中，设计了以下四组中断信号线：

INTA#、INTB#、INTC#、INTD# 一共四组中断信号线。

INTA#适合于一般单一功能(Single Function)的PCI装置，INTB#、INTC#、INTD#则适用于多重功能(Multi-functional)的PCI装置，有一点像是PCI世界中的中断信号中再去细分的次级中断代号。而大多数PCI适配卡，都是设计拉出INTA#作为中断处理的信号，来通知芯片组以及CPU做处理，此时其他INTB#、INTC#、INTD#信号无意义也不能使用。

这些装置会实际对映到芯片组上的PIRQ0、PIRQ1、PIRQ2、PIRQ3，BIOS以预先烧写到里面的IRQ绕线设置数据，来针对每个装置设置它与PIRQn的对映关系。而PIRQn最后会真实地对应到8259A PIC(可编程中断控制器)，去产生IRQ0～IRQ15的硬件中断信号。

而所谓的PCI IRQ Routing(PCI IRQ中断绕线设置)，就是将硬件(主板)上每个PCI装置、组件之间所使用的IRQ中断线路之间的关系，主板设计时对于IRQ中断信号线路的连接方式，如哪个芯片、哪个扩充槽使用到哪一组INTA#(或INTB#、INTC#、INTD#)，又连接到芯片组的哪个PIRQn等等，把它们的处理关系做成一个可以供BIOS以及操作系统能够理解、解读的参数

表格，这样才能让 BIOS 以及操作系统能够顺利处理、应付各种装置的中断而不至于相冲突。

事实上，如果这部系统 (或主板)的 PCI IRQ Routing 没有处理好，根本在开机 POST 阶段，系统就会停止(halt)住，而你会发现 POST Card 除错码停在 2Ah 或 38h 的位置。

常见的 PCI IRQ Routing 的设置方式

至于常见的PCI IRQ Riuting绕线方式又是如何？以下是一块以四组 PCI 扩充槽为蓝本，每个扩充槽的 INTA#/INTB#/INTC#/INTD# 以及跟芯片组、8259A PIC 中断控制器的连接情形 (如图6-3所示)。

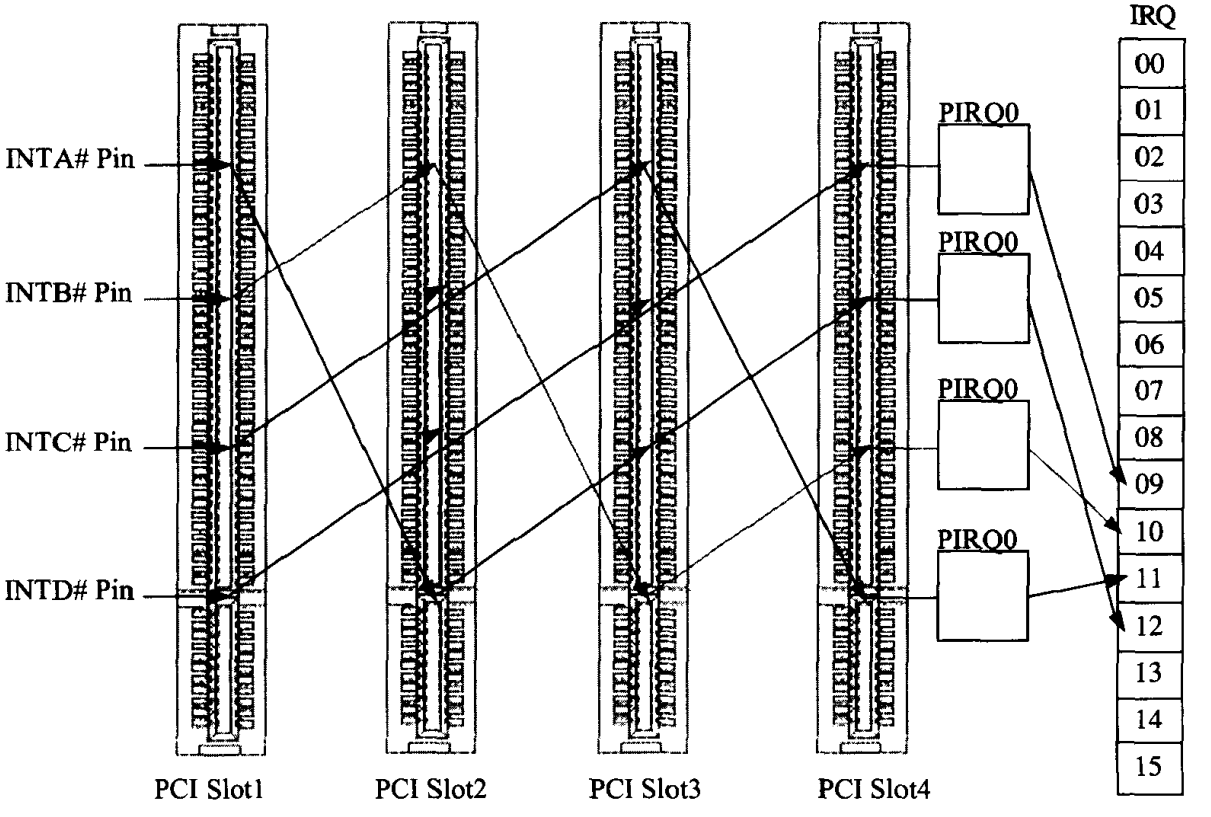


图6-3 芯片组的 PCI 装置 IRQ Routing 表格示意图

这样设计的用意，就是在四组 PCI Slot 都插上最常见到、只设计用到 INTA 来驱动中断信号的 PCI 适配卡，即使在四个 PCI 扩充槽都插上、插满了，这四块适配卡都能分配到不同且独一无二的系统 IRQ 号码，而不需要以共享的方式处理。

当然，PCI 装置或适配卡不见得都得用到中断信号的产生方式，来跟主板系统联系。像笔者手边的 PCI 接口的 POST/Debug Card，它只解读主板对 Port 80h 所送出的数据，不需要输出任何数据到主板，自然就不需要任何中断信号。

AMI BIOS 的 PCI IRQ Routing 绕线设置与表格形式

AMI BIOS 对于 PCI IRQ Routing 绕线设置的参数表格，是由一个叫做 OEMRPCI.ASM 来编译决定的，它是放在 OEM 厂商层底下的 OEMPORT 目录下。以笔者移植实作的 Leadtek 主板为例，其 PCI IRQ Routing表格就是位在\627.10\SIS530\OEM\DEMO1\OEMRPCI.ASM 里面。

AMI 已经利用宏指令针对 PCI IRQ Routing 项目设计了一个称为 pci_irq_entry 的数据定义宏指令，OEM 厂商只要使用 pci_irq_entry 这个指令，然后中间填入项目参数，有几个 PCI 装置就在 OEMRPCI.ASM 加几道 pci_irq_entry 就行了。以下是这个数据定义指令的格式说明：

Device Location	Int A# Pin	Int B# Pin	Int C# Pin	Int D# Pin	对应 Slot 号码	保留位组
PCI Bus DevNum Number (Shl 3)	Chipset Reg 对应的 IRQ位	Chipset Reg 对应的 IRQ位	Chipset Reg 对应的 IRQ位	Chipset Reg 对应的 IRQ位		

这个数据宏指令一共要填入六个项目的数值，以下是这六个项目的说明：

① PCI Bus Number (PCI 总线号码)

这个字段要输入 PCI Bus Number，PCI Bus 可以有两组或两组以上的 Bus，通过所谓的 PCI 跨桥接芯片(PCI Bridge)来相互连接。各位也许在服务器主板看过那种多达六组、八组 PCI 扩充槽，多半就是这种系统结构；而彼此之间是以 PCI Bus Number 来区分。隶属于第一组 PCI 控制器的 PCI 装置或扩充槽，用 PCI Bus 0，而隶属于第二组 PCI 控制器的 PCI 装置或扩充槽，则使用 PCI Bus 1 。在 PCI 的定义上，最多可达 32 组 PCI Bus 互连，这实在够强悍、够复杂了吧！

在一般最单纯的 PCI 总线系统，像常见到的台式计算机主板，即使是有双 CPU 的设计，由于 PCI 总线部分都是由北桥芯片的 PCI 控制器去控制，只有一组 PCI Bus 的设计，所以这个项目都是填 00h (第一组，而且为唯一的一组) 的机会居多。

有些有多组 PCI 插槽的超级服务器等级的主板上，有使用到扩展型 PCI 跨桥接芯片，因此提供不止一组以上的 PCI Bus，此时你要做 PCI IRQ Routing，就要看 On Board 的 PCI 装置 / 扩充槽是隶属于哪一组 PCI Bus 总线，然后填入对应的数值。

② Dev Num (Device Numbver 装置号码)

这是该 PCI 扩充槽或 PCI 装置的装置 ID 号码 (Device ID)，向左移三个位(乘以八)之后的数值。有关扩充槽与装置号码转换规则如下表：

PCI 扩充槽 ID 选择与装置号码查询表

ADsel	AD11	AD12	AD13	AD14	AD15	AD16	AD17	AD18	AD19	AD20	AD21
Device	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0Ah
Dev Num	00h	08h	10h	18h	20h	28h	30h	38h	40h	48h	50h
Adsel AD22	AD23	AD24	AD25	AD26	AD27	AD28	AD29	AD30	AD31		
Device	0Bh	0Ch	0Dh	0Eh	0Fh	10h	11h	12h	13h	14h	
Dev Num	58h	60h	68h	70h	78h	80h	88h	90h	98h	A0h	

Onboard 装置以该装置硬件线路对外拉出的 ADsel 号码，减去 11 之后的值为 Device 号码，Device 号码限制在 00～1Fh 之间。

③ Chipset Reg (IRQ 中断对映寄存器号码,也就是 PIRQn 的索引端口号码)

每一家芯片组的寄存器号码各有不同，这部分得详细参考该芯片组的技术文件以及 BIOS 编写指南。以 SIS530 芯片组为例，INT PinA 使用到 41h，INT PinB 使用到 42h，INTPinC 使用到 43h，INT PinD 使用到 44h，而 SiS530 Device 0001 又指定连到 62h；若没有连接到该组 INTx# 信号线，则填入 00。

④ BitMap of IRQs (指定可以分配使用的 IRQ 号码)

此字段由 16 个 bit 所组成，每个字段位设为 1，代表该字段所对映的 IRQ 号码可以加入分配使用；若设为 0 则禁止使用该 IRQ 号码。

例如我在这个字段填了 0DEB8h 这个十六进制数值，把 0DEB8h 以二进制表示如下：

	IRQ 15	IRQ 14	IRQ 13	IRQ 12	IRQ 11	IRQ 10	IRQ 09	IRQ 08	IRQ 07	IRQ 06	IRQ 05	IRQ 04	IRQ 03	IRQ 02	IRQ 01	IRQ 00
	1	1	0	1	1	1	1	0	1	0	1	1	1	0	0	0
bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

第 15，14，12，11，10，7，5，4，3 个位字段都设为 1，因此指定这个装置可以选用 IRQ15，IRQ14，IRQ12，IRQ11，IRQ10，IRQ7，IRQ5，IRQ4 与 IRQ3 这些中断，并通过 Plug and Play (PnP) 的分配方式配置一组专用的 IRQ，或者跟其他的装置一起 Share 。

⑤ Physical Slot Num 该装置所实际对映的 PCI 扩充槽号码数

这个字段用来说明这个装置实际连接在哪一个扩充槽上。数值从 1～6，因为目前台式机主板的 PCI 扩充槽最大数量也只有六只而已。

对于一些设计在主板上 (On Board)的 PCI 装置，如南北桥芯片、USB 控制器、音效芯片、IDE/Raid 或 SCSI 芯片等等，它们并不实际占用 PCI 扩充槽，这个字段就固定填 00。

⑥ Reserved byte 保留位

这一栏请填 00。

实际 OEMRPCI.ASM 的范例

笔者所实作的这块编号 WinFast 5300MA，SiS530 芯片组设计的丽台主板，有三组 PCI 扩充槽，也有 On Board PCI 音效芯片的设计。我们打开 \627.10\SIS530\OEM\DEMO1\OEMRPCI.ASM 文件来瞧瞧，它所定义的表格与参数项目。

这个 OEMRPCI.ASM 内部的 PCI IRQ Routing 表格一共有两份，一份是给 BIOS 刚 POST 阶段以及 Windows 95/98，Windows ME 等须借助 BIOS 的操作系统，而另一份则是给 Windows NT、Windows 2000 操作系统。

rth_pci_device_limit 这个长度为 byte 的数据域位，必须设置成这组绕线表格的项目总数，包含空白的字段；rth_pci_device_count 这个长度为 byte 的数据域位，必须设置成这组绕线表格的非空白项目的总数，任何空白的字段都不包含在内。

AMIBCP 也可以使用空白列，如果主板 OEM 厂商决定拉一组特别的 PCI 扩充槽在板子上的话，可以加入一组 pci_irq_entry <0, 0, 0, 0, 0, 0>，等 BIOS 编译完成之后，再通过像 AMIBCP 组态程序去 Edit IRQ Routing Table，详细做法请参考第 8 章。

文件名	OEMRPCI.ASM
路径	\627.10\SIS530\OEM\DEMO1\OEMPORT\OEMRPCI.ASM或 OEMPORT\OEMRPCI.ASM (相对于OEM厂商层)

程序开头有★表示此行原本跟上一行连在一起，因为排版的行列限制而切到下一行：

```
page  , 132
title  OEM RUNTIME HOOKS FOR PCI
;*****************************************************************************
;..................................... 版权声明。此部分叙述省略 .....................................
;*****************************************************************************
;-----------------------------------;
ifndef        RT32
              .286
else
              .386
endif
;-------------------------------- ;
      include rt.equ              ;
      include pci.equ             ;
      include devnode.equ         ;
      include mac32.mac           ;   加载相关的常数、宏指令定义文件
      include sis530.equ          ;
      include makeflag.equ        ;
;-------------------------------- ;
;    CODE   SEGMENT  ;
;-------------------------------- ;
ifndefRT32
      cgroup  group  _text
      _text segment para public 'CODE'
              assume cs:CGROUP
      .386
else
      .386
      cgroup group  _runtime32
      _runtime32 segment para public USE32 'CODE'
              assume cs:cgroup
```

```
endif

ifndefRT32
;---------------------------------------------------------------------------------------;
;                16 Bit Code Only             ;
;---------------------------------------------------------------------------------------;
;---------------------------------------- ;
;rth_pci_irq_routing_table                ;
;---------------------------------------- ;
;....本程序代码的批注部分省略 (有兴趣的读者请参考源代码原文批注) .......
;---------------------------------------------------------------------------------------;
    public rth_pci_device_limit        ;
    public rth_pci_device_count        ;   声明下列子程序、地址变量为公用性质
    public rth_pci_dedicated_irqs      ;   供日后编译、连结时让其他模块也能使用
    public rth_pci_irq_routing_table   ;

rth_pci_irq_routing_table_signature db '$IRT' ;供AMIBCP工具程序搜寻的标记
rth_pci_device_limit db 6                  ;下列表格内总共六个项目_
rth_pci_device_count      db 6             ;实际使用的装置表格最多也是六个
rth_pci_dedicated_irqs    dw 0000000000000000b    ;可供PCI分配的系统IRQ分
                                                  ;配位定义表bitn=1 IRQn
                                                  ;可参与分配，bitn=0 IRQn不
                                                  ;可参与分配
```

Device Location		IntA# Pin		Int B Pin		Int C Pin		Int D Pin		对应Slot号码	保留位组
PCI Bus Number	DevNum (Shl 3)	Chipset Reg	对应的IRQ位	Chipset Reg	对应的IRQ位	Chipset Reg	对应的IRQ位	Chipset Reg	对应的IRQ位		

pci_irq_entry<00h, 008h, 062h, 1101111010111000b, 000h, 0000000000000000b, 000h,
★0000000000000000b, 000h, 0000000000000000b, 0, 0>

;上述意思是Device 1 (SiS530 USB控制器)连接到PCI Bus 0,只使用INTA# Pin；索引寄存器为62h,可从IRQ 15,14,12～9,7,5～3中任选一组IRQ号码

Device Location		IntA# Pin		Int B Pin		Int C Pin		Int D Pin		对应Slot号码	保留位组
PCI Bus Number	DevNum (Shl 3)	Chipset Reg	对应的IRQ位	Chipset Reg	对应的IRQ位	Chipset Reg	对应的IRQ位	Chipset Reg	对应的IRQ位		

pci_irq_entry<00h,048h,041h,1101111010111000b,042h,1101111010111000b,043h,
★1101111010111000b,044h, 1101111010111000b, 1,0>
;上述意思是 Device 9 主板上的第一组 PCI 扩充槽，依次连接到 INTA#，
; INTB#，INTC#，INTD#，索引寄存器号码依次为 41h，42h，43h，44h，
; 可从 IRQ 15，14，12～9，7，5～3 中任选一组 IRQ 号码

Device Location		IntA# Pin		Int B Pin		Int C Pin		Int D Pin		对应 Slot 号码	保留位组
PCI Bus Number	DevNum (Shl 3)	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位		

pci_irq_entry<00h,050h,044h,1101111010111000b,041h,1101111010111000b,042h,
★1101111010111000b,043h, 1101111010111000b, 2,0>
;上述意思是 Device10 主板上的第二组 PCI 扩充槽，依次连接到 INTD#，
; INTA#，INTB#，INTC#，索引寄存器号码依次为 44h，41h，42h，43h，
; 可从 IRQ 15，14，12～9，7，5～3 中任选一组 IRQ 号码

Device Location		IntA# Pin		Int B Pin		Int C Pin		Int D Pin		对应 Slot 号码	保留位组
PCI Bus Number	DevNum (Shl 3)	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位		

pci_irq_entry<00h,058h,043h,1101111010111000b,044h,1101111010111000b,041h,
★1101111010111000b,042h, 1101111010111000b, 3,0>
;上述意思是 Device10 主板上的第二组 PCI 扩充槽，依次连接到 INTD#，
; INTA#，INTB#，INTC#，索引寄存器号码依序为 44h，41h，42h，43h，
; 可从 IRQ 15，14，12～9，7，5～3 中任选一组 IRQ 号码

Device Location		IntA# Pin		Int B Pin		Int C Pin		Int D Pin		对应 Slot 号码	保留位组
PCI Bus Number	DevNum (Shl 3)	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位	Chipset Reg	对应的 IRQ位		

pci_irq_entry<00h,060h,044h,1101111010111000b,000h,0000000000000000b,000h,
★0000000000000000b,000h, 0000000000000000b, 0,0>
;上述意思是 Device12(on board PCI 音效芯片)直接连接到 INTD#结束，索引寄
;存器号码 44h，可从 IRQ 15，14，12～9，7，5～3 中任选一组 IRQ 号码

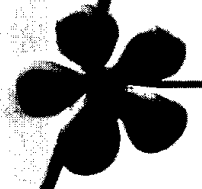
align 16
public ms_irq_route_table_sign

```
ms_irq_route_table_sign labelbyte      ;给 Windows NT/2000 的装置管理员分配 IRQ 的表格
db   "$PIR"                            ; 供 Windows NT/2000 装置管理员搜寻用的识别标记
ms_irq_route_table_ver     dw    0100h
ms_irq_route_table_size    dw    32+(6*16)
ms_irq_route_table_bus     db    00h
ms_irq_route_table_dev     db    S5595_PCI2ISA_DEV_FUNC_NUM
ms_irq_route_table_irq     dw    0h
ms_irq_route_table_id      dd    00081039h          ; SiS5595 南桥芯片的 PCI ID
ms_irq_route_table_mini    dd    0h
ms_irq_route_table_resd    db    11 dup (0)
ms_irq_route_table_cksm    db    0
;
pci_irq_entry < 00h, 008h, 062h, 1101111010111000b, 000h, 0000000000000000b,
★000h, 0000000000000000b, 000h, 0000000000000000b,    0,    0    >

pci_irq_entry < 00h, 010h, 041h, 1101111010111000b, 042h, 1101111010111000b,
★043h, 1101111010111000b, 044h, 1101111010111000b,    0,    0    >

pci_irq_entry < 00h, 048h, 041h, 1101111010111000b, 042h, 1101111010111000b,
★043h, 1101111010111000b, 044h, 1101111010111000b,    1,    0    >

pci_irq_entry < 00h, 050h, 044h, 1101111010111000b, 041h, 1101111010111000b,
★042h, 1101111010111000b, 043h, 1101111010111000b,    2,    0    >

pci_irq_entry < 00h, 058h, 043h, 1101111010111000b, 044h, 1101111010111000b,
★041h, 1101111010111000b, 042h, 1101111010111000b,    3,    0    >

pci_irq_entry < 00h, 060h, 044h, 1101111010111000b, 000h, 0000000000000000b,
」000h, 0000000000000000b, 000h, 0000000000000000b,    0,    0    >
;------------------------------------------------------------------ ;
COMMENT

endif
;-----------------------------------------------------------------;
;                    16/32 Bit Code                               ;
;-----------------------------------------------------------------;
```

```
;-------------------------------------------------------------------------    ;
;rth_pci_search_xlat                                                          ;
;-------------------------------------------------------------------------    ;
;供OEM厂商修改PCI装置的搜寻顺序(从0～1Fh或从1Fh找到0)。
;有些PCI装置的搜寻可能调整一下顺序，搜寻以及分配IRQ部分比较不
;会流于复杂而忙中出错!
.....本程序代码的批注部分省略(有兴趣读者请参考源代码原文批注)....
;Input:  BH = PCI bus number that is being searched                          ;
;       BL = PCI device search index "n" (in the range 0 - 1F)                ;
;                                                                            ;
;Output: BL = The "nth" PCI device number to search for (in the range 0 - 1F);
;                                                                            ;
;Destroys: Nothing                                                           ;
;-----------------------------------------------------------------------------;
        pubproc rth_pci_search_xlat
defproc rth_pci_search_xlat
;       sub     bl, 1Fh         ;Default is search Last-to-First
;       neg     bl              ;系统默认为从最后一个装置开始找起找到第一组
        ret
endproc rth_pci_search_xlat
;-----------------------------------------------------------------------------;
;*****************************************************************************;
;.....版权声明。此部分叙述省略.....
;*****************************************************************************;
ifndefRT32
        _text   ends
else
        _runtime32 ends
endif
        end
```

CHAPTER

7



BIOS 画面选项、默认值设计

在前面的章节曾提到过，BIOS 的Setup Server(设置服务端)以及Setup Client (设置客户端)两大功能模块，是组成整个 BIOS 组态设置的两大核心。Setup Client 是用户直接接触到的界面层 (Shell) ，它负责显示各种选项的画面、叙述，并且接受使用者按键或鼠标的输入，然后将必要的信息存储到 CMOS 内存；而 Setup Server 则负责从 CMOS 内存读取参数值并且做解释判断，然后执行对应的程序代码，来完成用户指定硬件项目的 Detect、Initializing、enable 或 Disable(硬件侦测、启动、打开 / 关闭)等动作。

本章要讨论的重点，是在 Setup Client 这个使用者能接触的模块部分，也就是它的界面设计方式。由于它的对外界面 / 呈现方式，光是从源代码那边修改仍不足以完成，必须借助其他工具程序才能成功！

7-1 AMI BIOS 的开机设置画面与选项简介

首先我们先回顾一下，AMI 提供了四种 BIOS 设置的操作界面形式：第一种是 HiFlex Setup(卷绕设置界面，如图 7-1、图 7-2 所示)，第二种是 Simple Setup 简单设置界面(如图7-3、图7-4所示)，第三种是 WinSetup(图形窗口设置界面，如图 7-5 所示)，第四种是 Easy Setup 轻松设置型界面 (如图 7-6 所示)。

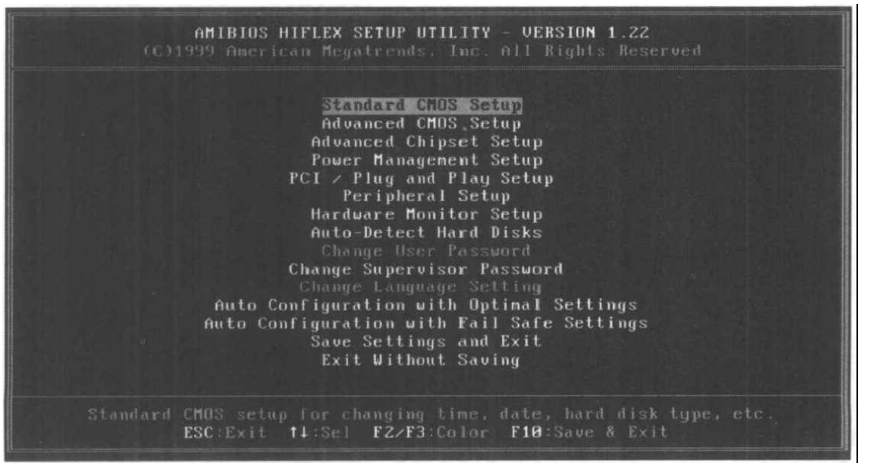


图7-1 HiFlex 界面的 AMI BIOS 设置开机画面-1

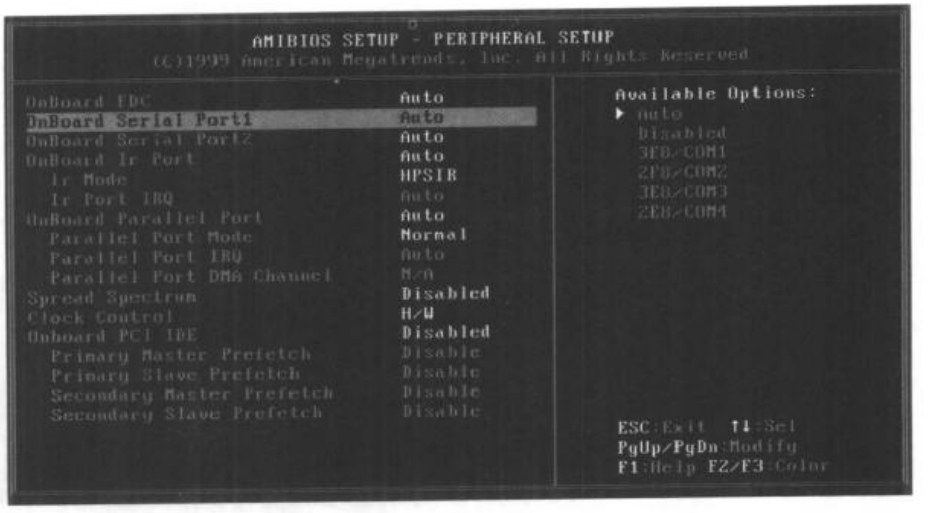


图7-2 HiFlex 界面的 AMI BIOS 设置开机画面-2

HiFlex 界面的 AMI BIOS，可以控制光条的移动方向只有从上而下或由下而上，一般在画面的左半部显示一部分项目，右半部则是显示可设置的数值范围；当控制光条从上一路往下选择时，隐藏未显示的选项从控制光条下方卷绕出来，适合于每个子选项的项目、文字叙述较多的 BIOS 画面配置。

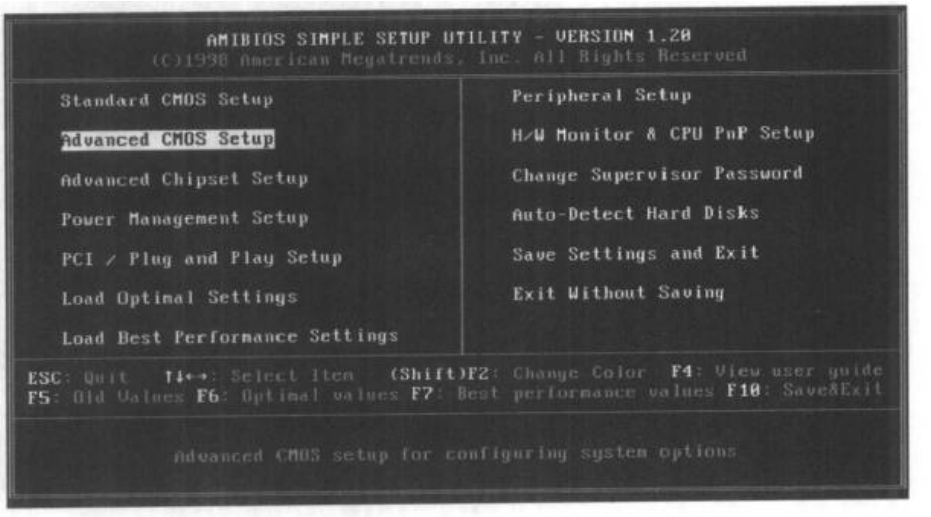


图7-3 Simple 界面的 AMI BIOS -1

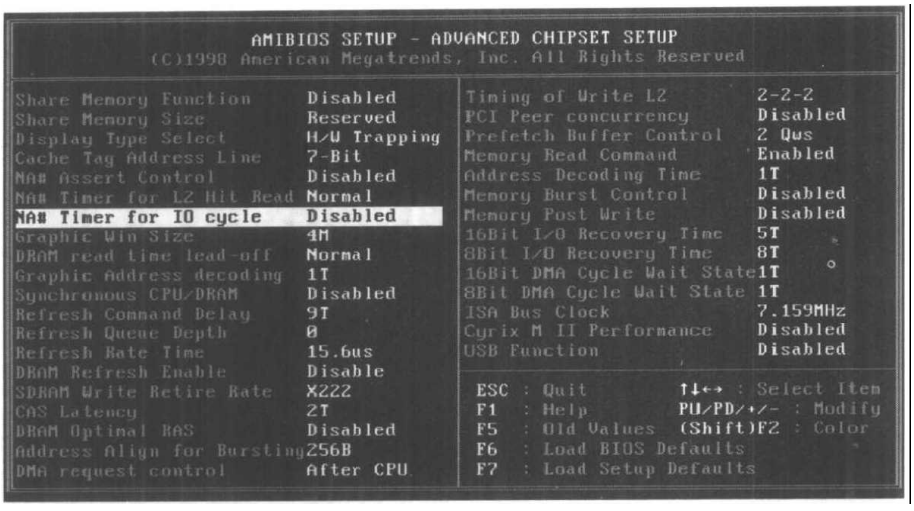


图7-4 Simple 界面的 AMI BIOS -2

至于 Simple 界面的 AMI BIOS，就跟你我经常见到的台式机的 AWARD BIOS 界面类似，所有项目从左到右、由上往下地摊开，控制光条移动的方向，至于每个项目可设置的数值是以 Page up、Page Down 或 +、- 键来调整的，这种界面比较适合于项目较少，或厂商已经精简过选项的 BIOS 画面配置。

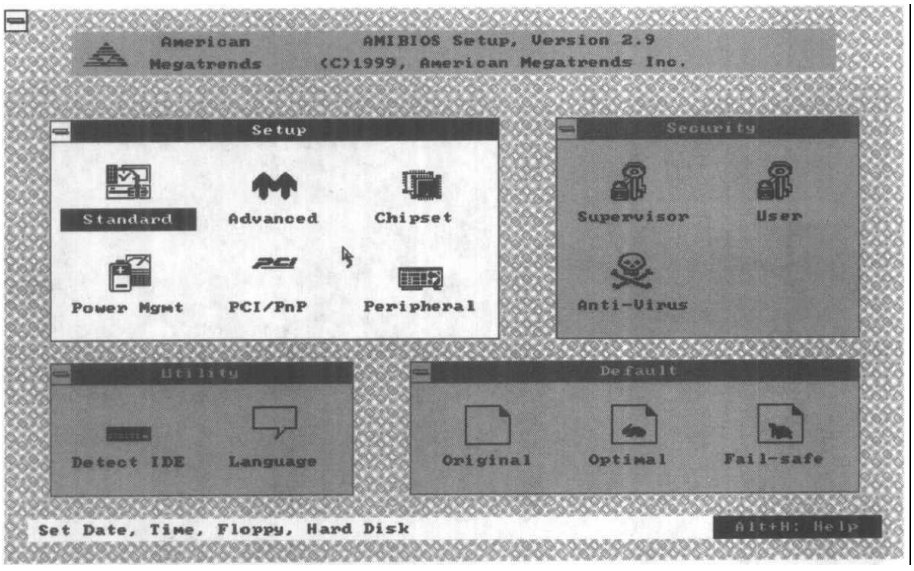


图7-5 Windows 界面的 AMI BIOS

Windows 设置界面则是以 VGA 字形发生器的技术，运用动态定义新字形来显现出图形的效果，而整个画面就类似 Windows 环境，每个主选项以一个图形化 Icon 来代表，除了支持以 PS/2 或 COM Port 的鼠标来移动光标、按钮来选择外，也可以按 [redacted] 切换作用窗口，用[redacted]、[redacted]来切换图像项目后按下[redacted]进入设置项目。

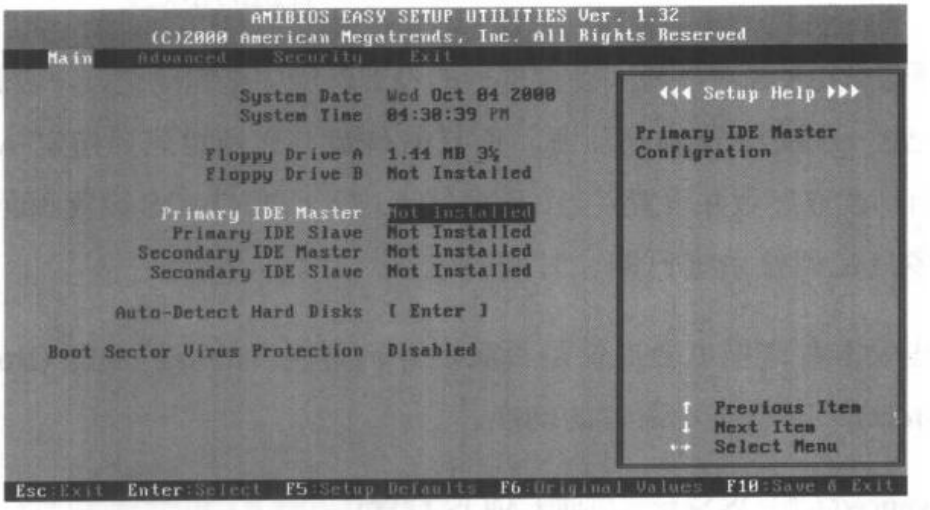


图7-6 EasySetup 界面的 AMI BIOS

至于 Easy Setup 界面就跟笔记本电脑上常出现的 Phoenix BIOS、SystemSoft SCU BIOS 的操作方式类似。就是左上方会呈现 Main、Advanced、Security 以及 Exit 四个主选项，用按 [redacted]、[redacted] 键来选择，切换到不同主选项时会显现不同的条列状细部选项，此时可按[redacted]或[redacted]、[redacted]键来选择要调整的字段；如果字段中有[redacted]的选项，则切换到这字段按下[redacted]后，会就进入下一层的选项画面。

无论是 HiFlex 或 Simple 界面，它的 BIOS 主设置画面，大致都会提供下列这些主选项目。

① **Standard CMOS Setup** (标准 CMOS 设置)

负责系统日期、时间的设置与记录，以及软驱类型(360K/720K/1.2M/1.44M/LS120) 的设置，另外也有 IDE 硬盘型号 / 容量的指定。

由于近代 BIOS 对软盘 / 硬盘型号的设置已经完全自动化，一般软盘驱动器已经一开始就能自动侦测，而 IDE 硬盘、光驱也只要指定 Auto，就会自动读取参数并设置妥当，因此读者进入标准 CMOS 设置画面的时候，大多只是调整系统日期、时间而已。

有些版本在这里也会提供是否激活开机型病毒的防护功能 (Boot Viurs Protect)，以及显示语言的切换。

② **Advanced CMOS Setup**(进阶 CMOS 设置)

关于开机装置的顺序、优先驱动的显卡(内置、PCI 或 AGP 显卡)、键盘反应速率、Boot Virus 防护功能、CPU L1/L2 快取功能开闭、处理器序号(Serial Number)开闭、 Microcode Update、BIOS Shadow 等项目。

③ **Advanced Chipset Setup**(进阶芯片组系统设置)

关于芯片组细部项目、功能与硬件特性参数的调整，如主存储器、影像内存时序、 CPU 种类、AGP 驱动模式等等。

④ **Power Management SETUP**(电源设置)

有关设置 ACPI，APM 节电模式的选项、进入节电模式的时间、要监控的外围装置(或 IRQ)、调制解调器 / 计时唤醒开机功能等。

⑤ **PCI/Plug and Play Setup**(PCI 外围与 PnP 设置)

关于 Plug and Play 与 OS 之间的搭配，IRQ、DMA 号码的指派与设置。

⑥ **Peripheral Setup**(外围零组件 / 装置设置)

针对主板上外围接口的开启、关闭，以及传输模式或 I/O 端口地址号码的指定。如主板内置的硬盘 IDE 接口、FDC 软盘接口、串行口的开启 / 关闭、IR 红外线传输端口的设置、打印端口的设置(标准 /EPP/ECP)。

若是主板具备 AC97 的音效 / 数据仿真功能，关于 AC97 Audio、AC97 Modem 功能的开启 / 关闭，也是在这个选项画面内。

⑦ **Hardware Monitor Setup** (硬件状态监控设置)

关于 CPU 的核心电压 (Vcore) 、外围电压 (3V、5V/12V) 、CPU 温度、散热风扇的马达转速的侦测。

有些主板强调可以通过软件方式来调整 CPU 的工作电压、倍频率、外频(Front-SideBus/Host Bus)等设置，多半也是归在这个画面；有些主板厂商，则是另外定义、设计一个称为 CPU Frequency/Voltage Control 的主选项。

⑧ **Auto-Detect Hard Disks**(IDE 硬盘自动侦测与设置)

自动侦测 IDE 装置的硬盘、CD-ROM 光驱的型号并存储到 CMOS。

⑨ **Auto Configuration with Optimal Settings**(HiFlex 界面)**Load Optimal Settings** (Simple 界面)(自动最佳化组态设置)

将主板厂商默认、预录的最佳化参数值存储到 CMOS。

⑩ **Auto Configuration with Fail Safe Settings**(HiFlex 界面)**Load Fail Safe Settings** (Simple 界面)(最佳安全开机组态设置)

将主板厂商默认、预录的最保守组态值，也就是最有可能排除一切障碍顺利开机的参数值存储到 CMOS。

⑪ **Change Supervisor Password** (更改超级用户密码)

设置 / 更改超级用户的密码。有些 BIOS 版本还多出 User Password(一般使用者)的密码设置选项。

⑫ **Save Settings and Exit**

存储设置到 CMOS 并离开设置画面。按下F10键也会有同样的效果。

⑬ **Exit Without Saving**

离开设置画面，设置结果不存储。按下ESC键也是同样的效果。

以上这 13 个选项，可能会因为各家主板厂商开发 BIOS 时，依自己的需要而对项目名称或项目总数做一些增减，但基本上主选项的文字大意应该都相差不多，设置项目的范围不出这十几项主选项范围内。

7-2 以 AMIBCP 调整开机默认值 / 隐蔽项目

随着年代的久远，计算机规格的不断演变，使得 BIOS 所支持的功能 / 规格越来越多，再加上处理器、芯片组、外围的差异，使得 BIOS 能提供的功能选择、硬件参数的调整与列举项目，变成非常的多元化，如果真要一一列举出来，在有限的 BIOS 画面与存储空间中，还真的是不胜枚举，不但列不完，用户要面对一大堆可调整的参数，想必也是迷惑不已。

AMI BIOS 当初在开发之际，似乎就想到了为这种多元化选项或参数的组态设置，预留了可以弹性调整、增减甚至预先设定的空间。厂商常有需要针对许多画面摆设、主选项、细选项做必要的精简；许多参数值也可以事先烧写在 BIOS 表格中，让使用者选择默认值后一切搞定，甚至是选项本身的文字叙述，都有可能去改写、增减或删除。

我们在第 9 章会介绍 AMIBCP (AMI BIOS Configuration Utility) BIOS 组态工具程序，它的功能非常强大，操作界面也是采用上下卷绕窗口配合控制光条的操控方式。我们可以通过 AMIBCP 这个程序，来帮我们调整开机的默认值，以及针对哪些项目做隐藏，甚至还可以将原本被隐藏的选项 / 叙述开启，重新显示于 BIOS 设置画面上。

此处以 1SIS5300.ROM 作为示范的 BIOS 影像文件名称。

① 首先我们在窗口环境下，进入“MS-DOS 模式”，然后在 DOS 提示符下键入 AMIBCP 1SIS5300.ROM，按 [redacted] 键，指定 AMIBCP 加载 1SIS5300.ROM 这个 BIOS 影像文件。

② 随后我们从 AMIBCP 主选项画面，选择第 6 行 Configure Setup DATA (设置文件组态)，如图 7-7 所示。

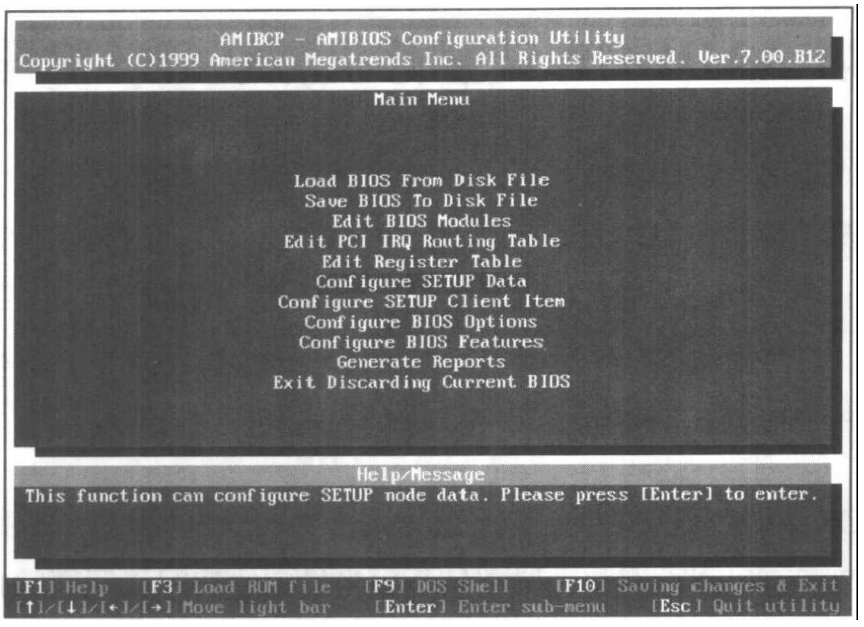


图 7-7　选择设置文件组态选项

③ 我们会进入如图7-8所示的画面，它有七个主要选择项目(上面介绍过)，我们以修改第一项Standard CMOS Setup(标准CMOS 设置)画面的默认值修改为例，这个画面有些限制，就是列出的选项不能隐藏，只能做默认值的调整，如图 7-9 所示。

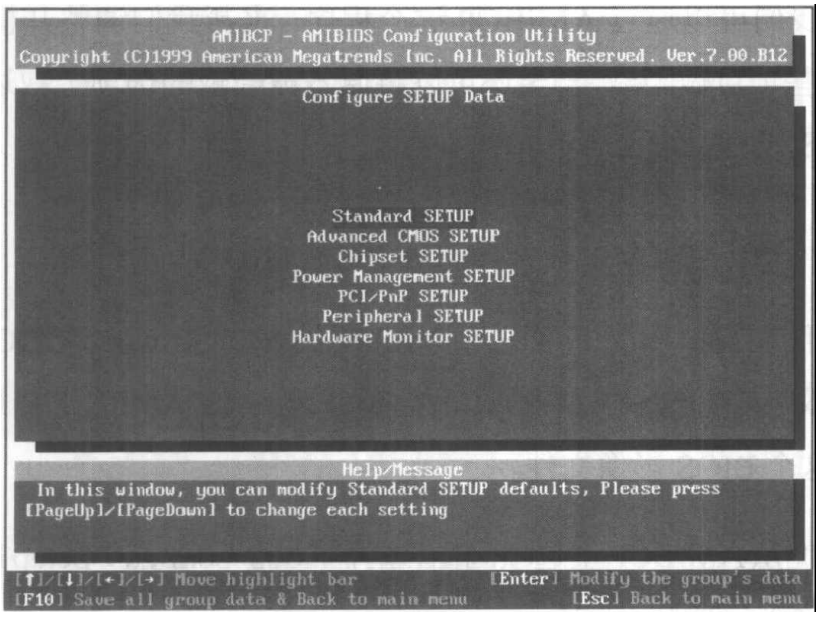


图 7-8　设置文件组态画面

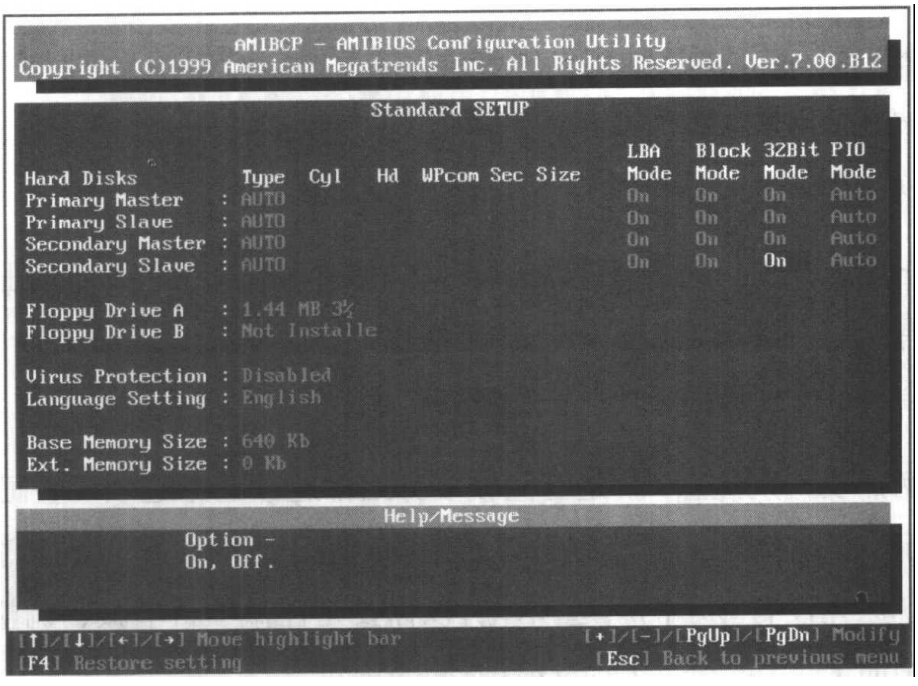


图 7-9　到希望指定开机默认值的字段，按[redacted]/[redacted]或[redacted]、[redacted]键来选择参数

接下来按[redacted]键回到图 7-8，再选择其他主要设定选项。此时我们选 Advanced CMOS Setup(进阶 CMOS 设置)画面，如图 7-10 所示。

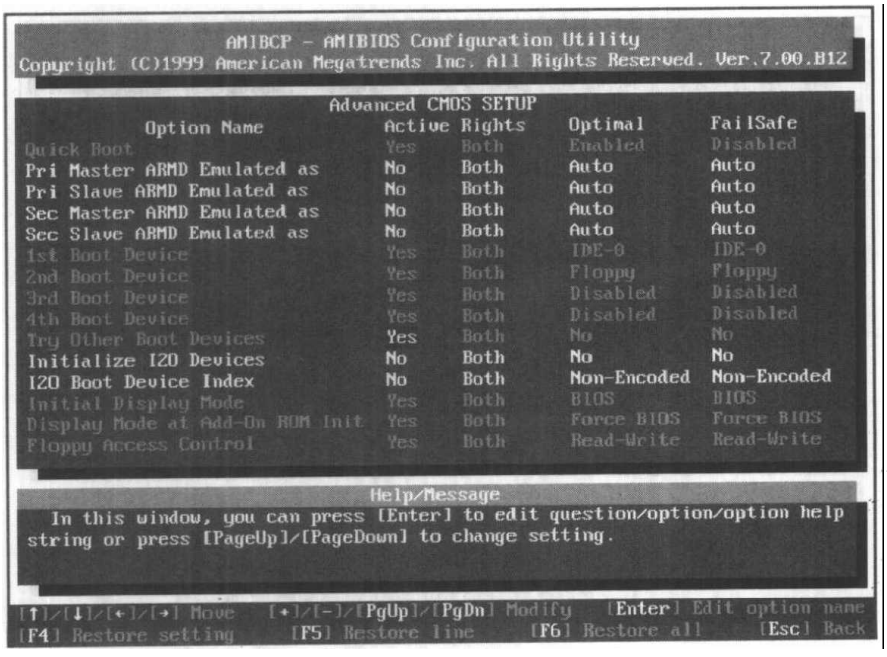


图 7-10　进阶 CMOS 设置

按[■]、[■]、[■]、[■]移动到你想要设置的字段。希望哪个项目在 BIOS 设置画面被显示出来，就在 Active 字段对该项目设成 Yes(有作用的)，若想隐藏某个选项，在 Active 字段选择 No。

Rights(显示权限)则是指定该选项只有在某个权限下才能显示的出来。这部分一共有二项：Supervisr(Supervisor 超级用户)模式，仅以 Supervisor 密码进入成功者会显示此项目；Both 则是超级用户或一般使用者都看得到这个选项。

有两点要特别注意，就是有些较多行细部选项的主选项设置画面，画面中只显示一部分选项，要等到你移动光标到最下方时，才会往上卷绕一行，同时出现原先没有显示的新选项，所以你要是在做项目增减、调整参数时，最好是画面上下卷一下，确定没有遗漏的项目。

以进入 Adanced Chipset Setup 画面为例，一开始屏幕只会显示如图7-11的画面，事实上下方还有一小部分没有显示出来，你必须将光标往下移动，隐藏在下方未显示的项目才会显示出来，如图7-12所示。

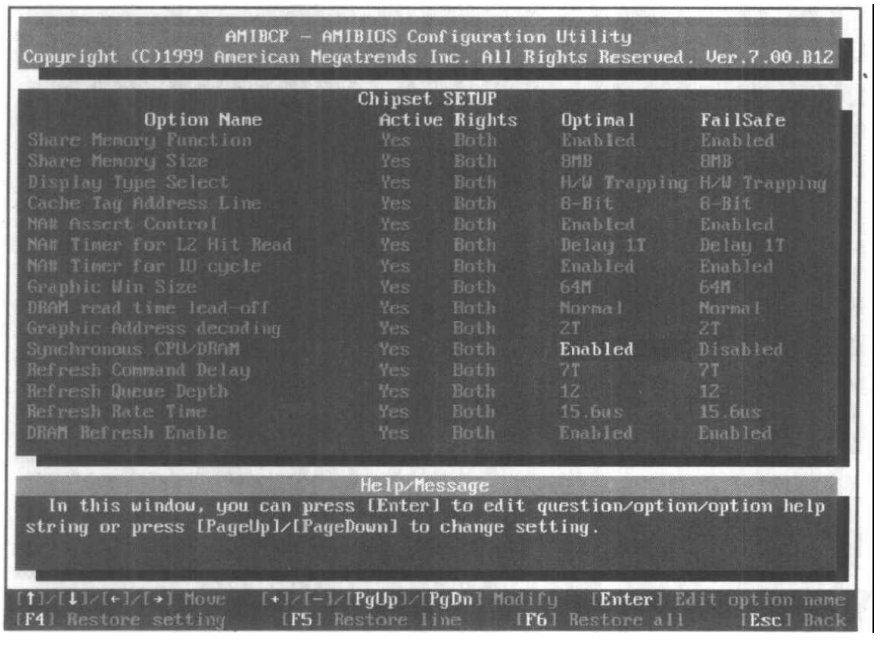


图 7-11　只有显示上半部选项

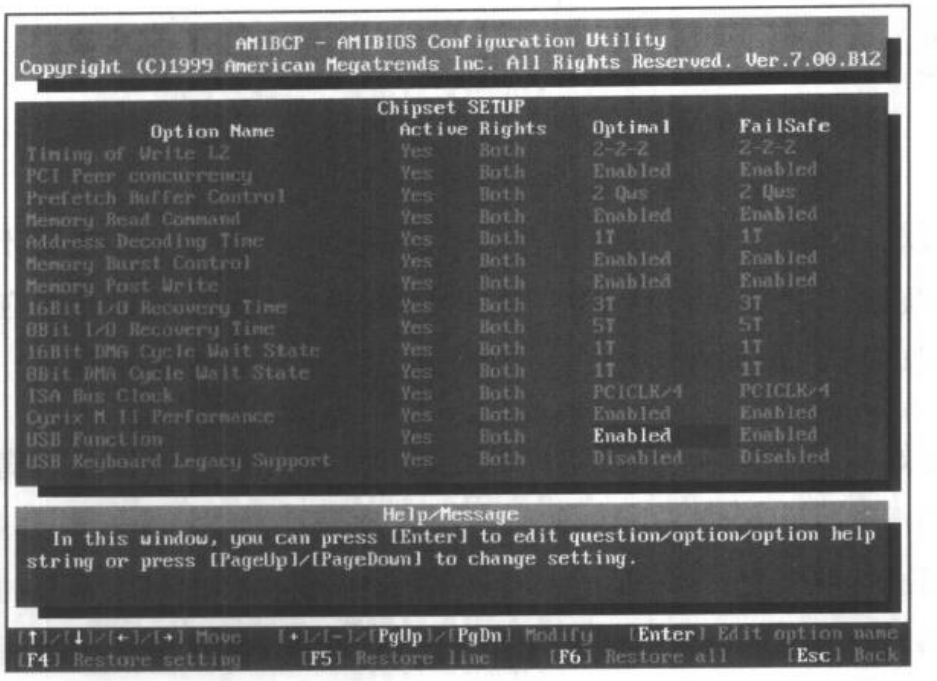


图7-12 下半部选项卷绕上来了

第二点则是你目前的设置，在按下[redacted]键回到主设置画面后，一定要记得按下[redacted]键，此时 AMIBCP 才会把更动过的选项调整值，重新保存到 1SIS5300.ROM 影像文件中。当然使用 AMIBCP 来调整 BIOS，是已经可以进步到针对某些项目做隐藏或默认值的调整，不过要是厂商或研发人员，已经加入新的芯片、新的电路，也想要新增加一些供使用者可以针对创新功能的选择(如知名的主板超频功能)，那又该怎么办呢？

难道我们得去动到 BIOS 的源代码 (.ASM)，想办法把新的选项控制程序码加进去，然后慢慢地修改、编译、除错，直到一切测试无误为止才算完成，这样做会不会太麻烦了些？

别急，后面的章节会介绍更方便、更符合人性的选项，甚至整个画面的定义方式。

7-3 AMI BIOS 的 Setup Script (.SSP) 语言的介绍

写过.ASM 汇编语言的读者，应该都知道用汇编语言来设计画面、选项是比较麻烦的，而且也是件很琐碎的事情。因此系统设计者应该把有限的研发时间、工夫花在刀口上，而不是去处理这些琐碎细节。

还好 AMI 当初在开发 BIOS 时，也许是经历动辄修改的痛苦，经过数年的演化，已经将 BIOS 的画面、选项设计独立出来另外作业。BIOS 的画面选项以及文字叙述，事实上是使用到一种 AMI 自家定义的虚拟设置描述语言，称为 Setup Script(.SSP)，它的语言形式跟一般高级语言相似，而且语法上十分好懂，系统开发者可以用文本处理器，来增加、减少一个选项提问与默认值。

为了让 BIOS 的各种选项、文字叙述可以随心所欲地定制与修改，AMI 发展了一套类似“语言产生器(Language Generator)”的工具程序——AMISSP.EXE，它的执行语法如下：

```
[C:\627.10\EXE]AMISSP
AMIBIOS Setup Script Processor
Version 2.07b (AMISSP Language Version 1)
Copyright (C) 1998 American Megatrends, Inc.
Usage: AMISSP <OutputPath> <InputFile> <InputFile> <InputFile> ...
```

以下是一个使用 AMISSP.EXE 的语法：

```
[C:\627.10\EXE]AMISSP C:\627.10\SIS530\OEM ..\OBJ\CORE.SSP
```

意思是指定 AMISSP 从目前的目录往上挪一层后切到 OBJ(也就是 C:\627.10\OBJ 目录)，然后读入 CORE.SSP 这个设置描述文件，翻译出来的结果输出到 C:\627.10\SIS530\OEM 目录底下。接下来我们开始介绍一下关于.SSP 语法。以下是个典型的.SSP 语法叙述规则。

```
Description = "ascii string"
LanguageVersion = 1.0

Screen (screen id)
    Question (question id)
      Text = "ascii string"
      Options = "optionstring1":DEFAULT, "optionstring2":POWERON, RESERVED, ...
      [Present = YES | NO]  >>> Default is <YES>
      [Supervisor = YES | NO]                    >>> Default is <NO>
      [ExternalFunction = function name] [OptionBits = n ]
                  [Default = xxxxh ]
                  [PowerOn = xxxxh ]
                  [Shaded = YES | NO]]          >>> Default is <NO>
      [Help = "helpstring1","helpstring2"]
      [ShadeWhen (parent question id) = "ascii string", "ascii string", ...]
      [OptionSet = "optionstring1","optionstring2",RESERVED, ...]
      [OptionSet = "optionstring3","optionstring4",RESERVED, ...]
      [OptionSet = "optionstring5","optionstring6",RESERVED, ...]

    EndQuestion
EndScreen
```

我们按次序介绍各关键词的定义与使用方式。

Description="*ascii string*"

定义：指定整个.SSP 的设置属性，归属于 *ascii string* 叙述所描述的功能集合。

目前能使用的 Description 有下列两种。

```
Description = "Generals setup items"      ←一般性项目的问题定义、设置等叙述
Description = "Peripheral setup items"    ← 针对 Peripheral 外围装置的设置叙述
```

而笔者参考了所有的.SSP 文件中，只有 CORE.SSP、IOC.SSP、IO1.SSP、IO2.SSP 以及 IO3.SSP 用到了 Description 的命令，CORE.SSP 是用“Generals setup items”，其余则都用到“Peripheral setup items”。但笔者无法从这些.SSP 的语法中看出有什么差异性，个人大致判断，做这样的描述有助于 AMISSP.EXE 在执行翻译作业时的顺利进行。

总之，如果你打开来修改的.SSP 描述文件中，有 Description 这一行叙述，请勿随意更动，以免造成无法顺利编译的后果。

LanguageVersion = 1.0

目前使用的.SSP 设置描述语言版本为 1.0，这是目前 AMISSP 描述语言编译程序所能支持的版本。

和上面一样，如果你打开的.SSP 有这一行，请勿随意更动，以免造成无法顺利编译的后果。

Question (*questionId*) EndQuestion

用途：指定从 Question (*questionid*) 行以下，一直到 EndQuestion 命令之前，这两行指令所涵盖的问题叙述句，均属这组 *questionid* 的管辖区。Question/EndQuestion 是最低层、最区域性的程序叙述，在屏幕上的显示单位通常只有一行。

Screen (*screenid*) 、EndScreen

用途：指定从 Screen (*screenid*) 此行以下一直到 EndScreen 命令之前，这两行指令所涵盖的语言叙述，均归属这组*ScreenId* 的管辖范畴。

Screen/EndScreen 是比 Question/EndQuestion 还高一层次的程序片段，它通常涵盖数个 Question/EndQuestion 的片段，一般是把这些同性质、属于同一个设置画面的几个 Question/EndQuestion 片段，以一个 Screen/EndScreen 作为较广域的叙述语言片段。

BeginDevice (*Devicestring*) 、EndDevice

用途：指定从 BeginDevice (*Devicestring*) 此行以下，一直到 EndDevice 命令之前，这两行指令所涵盖的语言叙述均归属这组 ScreenId 的管辖范畴。

这是比较会出现在偏外围装置设置（如上面介绍过的 Periphral setup items）的程序段落的定义命令，包含的层级比 Screen/EndScreen 还要高、还要广泛。以下是一个范例：

```
BeginDevice (SiS950FloppyController)        ←— 定义一个SiS950FloppyController的设备描述
                                                语言段落
Screen (SCR_PERIPHERAL)        ←— 定义归属于 SCR_PERIPHERAL 这组 ID
Question (Q_SIS950_FLOPPY_CONTROLLER)        ←— 定义一个 Q_SIS950_FLOPPY_
                                                CONTROLLER 的提问叙述
Text="OnBoard FDC"        ←— 显示“OnBoard FDC”的字符串
     Options =     "Auto":DEFAULT:POWERON,
                   "Disabled",
                   "Enabled"
                   "Secondary"
                                  定义一个Q_SIS950_FLOPPY_SWAP 的提问叙述
EndQuestion
Question (Q_SIS950_FLOPPY_SWAP)   ←┘
Text=" Floppy Drive Swap"         ←—显示“Floppy Drive Swap”的字符串
   ShadeWhen (Q_SIS950_FLOPPY_CONTROLLER) ="Disabled"
Options="Disabled":DEFAULT:POWERON,
                 "Enabled"
        EndQuestion

        EndScreen        ←— 目前 SCR_PERIPHERAL 这组 ID 叙述结束
EndDevice                ←— 整个SiS950FloppyController定义结束
```

以上有关 Question/EndQuestion、Screen/EndScreen 以及 Device/EndDevice，当 AMI 的 AMISSP.EXE 进行编译时，会依照同一种 ID 的 Question、Screen 以及 Device，做同层次程序片段的合并处理，还有后来居上的特性。

同层次 ID 合并处理的意思，是指例如两个分散的 Question 叙述，哪怕一个被写在 A.SSP，另一个写在 B.SSP，只要它们的 ScreenID 相同，则编译时会一起摆入同一组 ScreenID 的程序片段处理，视为同一个 ScreenID 下的程序片段。

至于后来居上的意思是指，例如同时有两组 Question，都指向同样一组 QuestionId 而形成重复的提问句时，即使一样是分布在不同的 SSP 文件，此时 AMISSP.EXE 并不会显示错误信息，而是以后面的 Question 来取代前面的 Question 的定义做处理。

StartSourceCode ("*filename.ext*")　EndStartSource

用途：将 StartSourceCode 到 EndStartSource 两个命令之间的所有字符串(前后双引号围住)，输出到 *filename.ext* 这个文件。

这个命令只有在CORE.SSP 以及ADVANCED.SSP 两个描述文件中可以看到。CORE.SSP 在目前的目录分别生成 SETUPDAT.ASM、SETUPEQU.ASM、SETUPEQU.EXT、SETUPMSG.ASM、SETUPMSG.EXT 与 SETUPRD.ASM 六个文件，供 MakeBIOS.BAT 做编译与链接的动作。

而 ADVANCED.SSP 一样会开启 SETUPDAT.ASM 这个.ASM 源代码，然后把一些变量给补上去。

很明显，如果读者对整个 AMI BIOS 的编译过程，以及每个关卡、过程的来龙去脉并不是十分了解的话，当你调出 CORE.SSP 以及 ADVANCED.SSP 有参考这些命令，以及后面输出的字符串叙述时，最好是不要随意改动，以免造成整套 BIOS 编译、链接的失败。

Include "*path\filename.ssp*"

用途：指定加载 *path\filename.ssp*，把指定的.SSP 描述文件加载后一同编译。路径与文件名字符串前后要加双引号。

这个命令可以在 CORE.SSP，IO.SSP 描述文件中发现。除了这些 AMI BIOS 开发者当初所指定的路径与文件不要随意改动之外，我们可以在自己的.SSP 中，善用 include 指令的特性，让整套 BIOS 的画面设计更具模块化、条件化，除错修改也将比较方便。

例如，最新的主板都具备 AC97 Audio & Modem 的功能，一般来说我们都是把提供 AC97 Audio & Modem 的提问叙述，给写在 IO2.SSP 中，不过当要实作其他没有 AC97 的主板 BIOS 时，又要把上述提问叙述给删除掉 (或者 remark 起来)；如果把 AC97 Audio & Modem 的提问叙述另外写一个 AC97.SSP 的描述文件，还需要在 IO2.SSP 加一行 include 命令如下。

```
Include "oemchip.mod\ac97.ssp"
```

整个 BIOS 的 SSP 描述文件就会更模块化、弹性化，甚至读者若肯下一些苦功，也可以运用条件判断式，去修改 OEM.MAK，让 Makebios.bat 能够视 AC97 有没有存在而自动加入 AC97.SSP 的描述文件的编译；对于以少数人力要维护旗下众多主板产品的研发人员而言，可以省下不少反复修改、除错的时间。

TEXT = "ascii string"

用途：显示一段“*ascii string*”的文字信息。

例如在一个问题叙述中，出现 TEXT=“CPU Front Side Bus”，则会在执行到这段命令叙述时，在 BIOS 画面显示出“CPU Front Side Bus”的文字。

Options="*optionstring1*"，"*optionstring2*",...

用途：列举可选择的选项字符串并显示出来。要显示的字符串必须在头尾各加一个双引号 (")，字符串与字符串之间以逗号做区隔，最后一个选项字符串则不需要加逗号。

DEFAULT POWERON RESERVED

用途：指定某个 Optionstring 为默认值 (DEFAULT) 或开机时必须先行填入的一个数值。

例如笔者想设计一个显示芯片的影像内存调整选项，预定可以调整的内存大小是 2MB、4MB、8MB、16MB、32MB，8MB 为最佳默认值，可以做下列的叙述。

```
Options="2MB ", "4MB ", "8MB ":DEFAULT, "16MB", "32MB"
```

如果选项文字叙述中有 RESERVED，表示这个选项预留、保留，待日后扩充选项字符串时所使用；当然有时是为了对齐某个选项字符串时使用。例如上述影像内存提问叙述，已经写好了.SSP，但今天进了一批主板芯片组，厂商告知它影像内存调成 4MB 会有问题，但其他容量下一切正常，因为商机的因素，上头告知要以这批芯片组来设计主板出货，此时就可以打开那组.SSP 文件，把上述的叙述修改如下。

```
Options="2MB ", RESERVED , "8MB ":DEFAULT, "16MB", "32MB"
```

Optionset="*optionstring1*","*optionstring2*",.....Optionset "*optionstring3*", "*optionstring4*",..... .....

用途：当你要提供的选项字符串相当多，而一行显然装不下去时，可以改用 Optionset=的叙述，每一行开头指定 Optionset，代表这一行也是选项字符串，紧接着上一行的选项字符串，成为整个选项数组。

使用 Optionset 有个限制，就是选项字符串的后面无法加上 DEFAULT 或 POWERON来指定哪个选项为默认值或开机设定值。

Present=YES|NO

用途：指定这行叙述要不要显示出来。AMISSP.EXE 的默认值均为 YES。

设计这个 Present 命令的用意，在于实作某系列主板 BIOS 时，原本某些功能／选择是提供的，因为某些缘故得先暂时关掉时，可以用 Present 命令来控制整个提问叙述句要不要显示出来。这项功能也可保留某个不知道该存还是要废的选项，日后可通过 AMIBCP 这个工具程序来设置显示或不显示。

Supervisor=YES|NO

用途：设置这个提问叙述句是否仅在 Supervisor 模式才会显示出来。默认值为 NO。

Help="*helpstring1*","*helpstring2*","*helpstring3*".....

用途：设置相对应于“*optionstring1*”,“*optionstring2*”,“*optionstring3*”... 选项文字的辅助说明字符串。辅助说明的字符串数目要跟选项字符串数目一致。

Shaded = YES | NO

用途：将这个提问叙述句设置成“纯列举模式”，届时只会在 BIOS 画面列出较低亮度的文字，无法做选项的输入与设置。默认值为 No。

ShadeWhen (*parent questionid*) = "*ascii string*", "*ascii string*", ...

用途：当上一层提问选项 ID 的字符串，符合某种字符串条件时，整行提问叙述变成低亮度的纯列举文字显示，不提供做选项设置，可作为选项之间的连锁互动的用途。

我们常看到BIOS的某些选项与选项之间，有彼此相互关联、互锁的机制。例如在一些可以调整 CPU 工作电压、外频与内频的选项中，如果我们想设计当使用者把CPU 调整的功能打开时，才会以高亮显示其他的选项，同时开放使用者来调整的连锁功能，我们可以做如下的设计。

```
Question (Q_CPU_ADJUST)
  TEXT = "CPU Adjust:"
  options = "Auto", "Manual"
EndQuestion

Question (Q_CPU_VOLT_SET)
  TEXT = "CPU Voltage:"
  Options="1.5V":DEFAULT,"1.55V","1.60V","1.65V","1.70V","1.75V","1.80V","1.85V"
    ShadeWhen (Q_CPU_adjust) = "Auto"
EndQuestion

Question (Q_CPU_FSB_SET)
  TEXT = "CPU FSB Clock:"
  Optionset="66MHz","100MHz","105MHz","110MHz","115MHz","120MHz","125MHz"
  Optionset = "130MHz","133MHz","140MHz","145MHz","150MHz"
    ShadeWhen (Q_CPU_adjust) = "Auto"
EndQuestion
```

以上定义了Q_CPU_ADJUST、Q_CPU_VOLT_SET以及Q_CPU_FSB_SET，当Q_CPU_VOLT_SET以及Q_CPU_FSB_SET侦测到Q_CPU_ADJUST被选择Auto时，这两个提问叙述只会以低亮度纯列举显示，而且无法做选项的输入与调整；只有在Q_CPU_ADJUST被设置成Manual时，此时Q_CPU_VOLT_SET以及Q_CPU_FSB_SET这两个提问叙述，选项字段才会以高亮文字模式显示，同时才能做选项的调整与设置。

String (*stringid*) 、EndString

用途：指定某个 (*Stringid*) 的字符串变量。

以下面的例子为例。

```
String (STR_ITE8661F_IRDA)
        Text="NA"
EndString
```

指定STR_ITE8661F_IRDA这个字符串变量的内容为“NA”，届时若显示STR_ITE8661F_IRDA这个字符串变量时，就会显示NA这段文字。

以上的.SSP语言设置，从选项字符串的显示、选项的切换，都已经由AMISSP.EXE帮你处理妥当，它们的参数都存放在CMOS内存内，除了已经公开保留的位置以外，许多新设计的选项变量，只要不是字符串变量，事实上都是动用并存放到CMOS内存位置。在稍后我们会有进一步的说明。

ExternalFunction = *function name*

用途：定义要跳越的子程序进入点的卷标名称，调用的子程序名称则位于 EXTFUNC.ASM 里面。

有时这个提问叙述在做选择时，必须先去执行一小段 BIOS 其他程序片段，返回之后才能继续提供其他项目的选择时，就得定义 External Function 这个命令。

如取得 SDRAM 的 SPD 时序参数，选择硬盘型号为 Auto 时，要先调用 BIOS 其他模块中侦测 IDE 硬盘的子程序码，影像内存的调整，还有读取 CPU 散热风扇的马达转速, 或者 CPU 的核心电压侦测值 (通过 LM75/LM78) 等，此时可以定义这个提问叙述，指定要显示这个提问文句之前，先去调用位于 EXTFUNC.ASM 中的哪个子程序进入点。

EXTFUNC.ASM 位于..\..\SYS\目录下，有些会附在 OEMCHIP 目录下，如果 External 指令所指定的子程序进入点，并没有在 EXTFUNC.ASM中找到，此时整个 BIOS 的 Assembling、Linking 就会发生错误。

BIOS 的研发 / 维护工程师，可以从目前的计划目录，往上切到..\..\SYS 目录，把EXTFUNC.ASM 拷贝一份到目前计划目录下的 OEMCHIP.MOD 目录下，然后针对自己设计主板的需求来作修改。

LocateCmos = *xxh* : *offsetbit*

用途：指定这个提问叙述的变量存在 CMOS 的绝对位置 xxh (00～07Fh, 最多 128 组) 地址，从第 offset bit 处的字段开始 (0～7) 。

一般而言，AMISSP.EXE 会将你定义的提问变量，以“动态配置”、“随选随用”的方式，选择一个尚未使用到的CMOS地址与偏移位字段来存放、使用。

不过 CMOS 内存的某些区域，有许多已经固定位置 / 位字段的变量被 BIOS 所固定占用，如 CMOS 系统时间、日期、系统设备位等相关系统信息，或者希望某些定义的变量能够有些固定字段，以方便做日后的维护。BIOS 研发者就可以善用这个 LocateCmos 的命令，把这个变量定义成强制使用某个 CMOS 绝对位置，并且指定使用哪几个位字段。

举例说明。

```
Question (Q_TEST_1)
      LocateCmos = 11h:0
      Text = "TEST QUESTION 1"
      Options = "Disabled":DEFAULT:POWERON
            "Enabled"

  EndQuestion

  Question (Q_TEST_2)
      LocateCmos = 14h:2
      Text = "TEST QUESTION 2"
      Optionbits = 2

  EndQuestion
```

以上分别定义Q_TEST_1这个变量，强制使用 CMOS 索引地址 011h 处的内存，从 bit0 字段开始使用一个 byte 的长度；定义Q_TEST_2这个变量强制使用 CMOS 索引地址 014h 处的内存，从bit2字段开始占用两个位 (Bit2～Bit3) 字段宽度。

OptionBits = n

用途：指定使用最大的选项字段的位数。N ≤ 16，最大可提供 2^N 个选项选择。

一般定义提问变量时，若 OptionBits 未指定，则默认值使用一个 byte (N=8) 。

若 OptionBits = 6，表示这个提问叙述的变量，它的字段只有六个位，因此有 2^6=32，有 32 种数值可供选择。选择好的数值与存放位置，会经过 AMISSP.EXE 计算，直接生成对应的.ASM 程序代码片段，并且存储于 CMOS 内存内。

当 N>8 时，代表这个定义变量使用到两个以上的字节，此时 AMISSP.EXE 事实上会生成两个变量：低字节的 questionid 以及高字节的 questionid_1，而 Mask 屏蔽值也会有两组。以下列为例。

```
Question (Q_TEST_WORD)
  Text = "TEST QUESTION WORD"
  Optionbits = 15
EndQuestion
```

届时会在翻译的过程中，在 EXTFUNC.ASM 生成对应的低字节变量 Q_TEST_WORD(bit0～bit7)，屏蔽值为 0FFh；以及高字节变量 Q_TEST_WORD_1 (bit8~14)，屏蔽值则为 07Fh。读者可以在编译之后，在目前计划目录下的..\..\ROM 目录下，找出 SSP.REP 来，稍后我们会再仔细介绍这个文件。

Default = xxxxh 、PowerOn = xxxxh

用途：指定默认值或开机值为 xxxxh，单位为 16 进制，但不可超过 Optionbits 所能表示的数值范围。

以下是本书所附光盘，位于 OEMCHIP.MOD 目录的 CHIPSET.SSP 以及 EXTFUNC.ASM 两个文件，我们拿其中的部分片段做说明。

文件：目前计划目录(Projectname) \oemchip.mod\CHIPSET.SSP

```
String (STR_1MB)
        Text  = "1MB"
EndString

String (STR_2MB)
        Text  = "2MB"
EndString

String (STR_4MB)
        Text  = "4MB"
EndString

String (STR_8MB)
        Text  = "8MB"
EndString

Screen (SCR_CHIPSET)

                Question (Q_SHARE_MEMORY)
                        Text = "Share Memory Function"
                        Options = "Disabled"
                                        "Enabled":DEFAULT:POWERON
                        Present = Yes
                EndQuestion
```

```
Question (Q_SHARE_MEMORY_SIZE)
                Text = "Share Memory Size"
                ExternalFunction = share_memory
                OptionBits = 2
                Default = 03h
                PowerOn = 03h
                Present = Yes
EndQuestion
```

文件：Goemchip.mod\extfunc.asm，从第 260 列开始

```
;-------------------------------------------------------------------------------;;
;  Share memory size                                                            ;;
;-------------------------------------------------------------------------------;;

SHARE_MEMORY_zlist label    byte
db      STR_RESERVED,STR_2MB,STR_4MB,STR_8MB,STR_END
NA_zlist label byte
db      STR_IT8661F_NA,STR_IT8661F_NA,STR_IT8661F_NA,STR_IT8661F_NA;db
STR_WIN877_NA,STR_WIN877_NA,STR_WIN877_NA,STR_WIN877_NA
Share_memory    proc near
            mov   ax,cs
            cmp   ax,0f000h
            mov   ah,2
            mov   bx,offset cgroup:SHARE_MEMORY_zlist
            jnz   short setting_done
            mov   al,Q_SHARE_MEMORY; 取得 Q_SHARE_MEMORY 这个变量 call
            get_cmos_buffer_item   ; 调用 get_cmos_buffer_item 子程序
            jnz   setting_done
            mov   ah,3
            mov   bx,offset cgroup:NA_zlist
```

```
setting_done:
        mov   dx,cs
        retf
Share_memory  endp
```

以下省略

而真正这些在.SSP 所定义的 Q_SHARE_MEMORY，以及其他的变量所对应的 CMOS 位置，可以从..\..\ROM\目录下的 SSP.REP 中找出来，这是 因为 AMISSP.EXE 每次编译*.SSP 之后，会把所有使用到 CMOS 的索引地址，以及索引屏蔽字节，以列表的方式记录在 SSP.REP 这个文件。如果是较早版本的 AMISSP.EXE，会生成下列列数较少，信息比较简略的 SSP.REP。

```
;****************************************************************************
;****************************************************************************
;**                                                                        **
;**   (C) Copyright 1985-1998, American Megatrends, Inc.                   **
;**                                                                        **
;**                        All Rights Reserved.                            **
;**                                                                        **
;**              6145-F Northbelt Pkwy, Norcross, GA 30071                 **
;**                                                                        **
;**                   Phone  (770) -246-8600                               **
;**                                                                        **
;****************************************************************************
;****************************************************************************
AMIBIOS Setup Script Processor
Version 2.02b (AMISSP Language Version 1)
Copyright (c) 1998 American Megatrends, Inc.

Input files:
c:\masm611\tmp\mme03857. (Peripheral setup items)
```

CMOS Question	(索引位置) Reg.	(屏蔽值) Mask
Q_QUICK_BOOT	13h	80h
Q_PM_EMULATION_TYPE	52h	03h
Q_PS_EMULATION_TYPE	52h	0Ch
Q_SM_EMULATION_TYPE	52h	30h
Q_SS_EMULATION_TYPE	52h	C0h
Q_FIRST_BOOT_DRIVE	53h	0Fh
Q_SECOND_BOOT_DRIVE	53h	F0h
Q_THIRD_BOOT_DRIVE	54h	0Fh
Q_FOURTH_BOOT_DRIVE	54h	F0h
Q_REMAINING_BOOT_DRIVES	55h	01h
Q_INIT_I2O_DEVICE	55h	02h
Q_INDEX_I2O_BOOT_DRIVE	56h	FFh
Q_INITIAL_DISPLAY	55h	04h
Q_ADDON_ROM_DISPLAY	55h	08h
Q_FDD_ACCESS_CONTROL	55h	10h
Q_HDD_ACCESS_CONTROL	55h	20h
Q_SMART_IDE	55h	40h
Q_BOOTUP_SPEED	11h	80h
Q_NUM_LOCK	11h	01h
Q_TURBO_SWITCH	13h	20h
Q_FLOPPY_SWAP	11h	20h
Q_FLOPPY_SEEK	11h	10h
Q_MOUSE_SUPPORT	29h	40h
Q_TYPEMATIC_RATE	11h	04h
Q_KEYBOARD_PRESENT	14h	04h
Q_DISPLAY_TYPE	14h	38h
Q_PASSWORD_CHECK	29h	10h
Q_PARITY_ERROR	29h	08h
Q_OS2_COMP_MODE	29h	80h
Q_WAIT_FOR_F1	11h	02h
Q_DISPLAY_HIT_DEL	11h	08h

Q_INTERNAL_CACHE	57h	03h
Q_EXTERNAL_CACHE	57h	0Ch
Q_CACHE_BIOS	55h	80h
Q_C0000_SHADOW	57h	30h
Q_C8000_SHADOW	57h	C0h
Q_CC000_SHADOW	58h	03h
Q_D0000_SHADOW	58h	0Ch
Q_D4000_SHADOW	58h	30h
Q_D8000_SHADOW	58h	C0h
Q_DC000_SHADOW	59h	03h
Q_SHARE_MEMORY	59h	04h
Q_SHARE_MEMORY_SIZE	59h	18h
Q_MONITOR_DEVICE	59h	E0h
Q_CACHE_TAG	5Ah	01h
Q_NA	5Ah	02h
Q_NA_L2_HIT	5Ah	04h
Q_NA_IO_CYCYE	5Ah	08h
Q_GRAPHIC_WIN	5Ah	70h
Q_DRAM_SYNCHRONOUS	5Ah	80h
Q_GRAPWIN_ADD_DECODE	5Bh	01h
Q_SYNCHRONOUS_CPU_DRAM	5Bh	02h
Q_RAS_PULSE_WIDTH	5Bh	0Ch
Q_REFRESH_DEPTH	5Bh	30h
Q_REFRESH_TIME	5Bh	C0h
Q_REFRESH_ENABLE	5Ch	01h
Q_SDRAM_WR_RETIRE	5Ch	02h
Q_CAS_LATENCY	5Ch	04h
Q_SDRAM_ROWHIT	5Ch	08h
Q_MAX_BURSTABLE	5Ch	70h
Q_DMA_REQ_CNT	5Ch	80h
Q_TIME_WRITE_L2	5Dh	01h
Q_CONCURRENCY_MODE	5Dh	02h
Q_TRDY_ASSERT_TIME	5Dh	04h
Q_READ_PREFETCH	5Dh	08h
Q_ADD_DECODE_TIME	5Dh	10h

Q_MEM_BURST_CNT	5Dh	20h
Q_MEM_POST_WRITE	5Dh	40h
Q_16BIT_IO_CYCLE	5Eh	03h
Q_8BIT_IO_CYCLE	5Eh	0Ch
Q_16BIT_DMA_CYCLE	5Eh	30h
Q_8BIT_DMA_CYCLE	5Eh	C0h
Q_ISA_BUS_CLOCK	5Fh	03h
Q_CYRIX_MII_PERFORMANCE	5Dh	80h
Q_USB_FUNCTION_ENABLE	5Fh	04h
Q_USB_LEGACY_ENABLE	5Fh	08h
Q_ACPI_OS	5Fh	10h
Q_POWER_MANAGEMENT	5Fh	20h
Q_GREEN_PC_MONITOR_STATE	5Fh	C0h
Q_VIDEO_POWER_DOWN	60h	03h
Q_IDE_POWER_DOWN	60h	0Ch
Q_IDE_TIME_OUT	60h	F0h
Q_STANDBY_TIME_OUT	61h	0Fh
Q_SUSPEND_TIME_OUT	61h	F0h
Q_SLOW_CLOCK	62h	07h
Q_MODEM_IRQ	62h	78h
Q_VIDEO_ACCESS	63h	03h
Q_IRQ3_PM_MONITOR	63h	0Ch
Q_IRQ4_PM_MONITOR	63h	30h
Q_IRQ5_PM_MONITOR	63h	C0h
Q_IRQ7_PM_MONITOR	64h	03h
Q_IRQ9_PM_MONITOR	64h	0Ch
Q_IRQ10_PM_MONITOR	64h	30h
Q_IRQ11_PM_MONITOR	64h	C0h
Q_IRQ13_PM_MONITOR	65h	03h
Q_IRQ14_PM_MONITOR	65h	0Ch
Q_IRQ15_PM_MONITOR	65h	30h
Q_POWER_BUTTON	62h	80h
Q_HOTKEY	65h	40h
Q_KEYBOARD_PASSWORD	65h	80h
Q_RING_ON	66h	03h

Q_RESUME_BY_RTC	66h	04h
Q_RESUME_BY_RTC_DATE	66h	F8h
Q_RESUME_BY_RTC_HOUR	67h	1Fh
Q_RESUME_BY_RTC_MINUTE	68h	3Fh
Q_RESUME_BY_RTC_SECOND	69h	3Fh
Q_PNP_AWARE_OS	67h	20h
Q_CLEAR_NVRAM	67h	40h
Q_ON_BOARD_DEVICE	67h	80h
Q_PCI_LATENCY_TIMER	6Ah	07h
Q_PRI_DISPLAY_CARD	68h	40h
Q_PCI_PALETTE_SNOOP	68h	80h
Q_PCI_VGA_IRQ	69h	40h
Q_PCI_IDE_BUS_MASTER	69h	80h
Q_PCI_OFFBOARD_IDE	6Ah	38h
Q_PCI_OFFBOARD_IDE_PRI_IRQ	6Bh	07h
Q_PCI_OFFBOARD_IDE_SEC_IRQ	6Bh	38h
Q_PCI_SLOT1_IRQ	6Ch	0Fh
Q_PCI_SLOT2_IRQ	6Ch	F0h
Q_PCI_SLOT3_IRQ	6Dh	0Fh
Q_PCI_SLOT4_IRQ	6Dh	F0h
Q_DMA0_USED_BY_ISA	6Ah	40h
Q_DMA1_USED_BY_ISA	6Ah	80h
Q_DMA3_USED_BY_ISA	6Bh	40h
Q_DMA5_USED_BY_ISA	6Bh	80h
Q_DMA6_USED_BY_ISA	6Eh	01h
Q_DMA7_USED_BY_ISA	6Eh	02h
Q_IRQ3_USED_BY_ISA	6Eh	04h
Q_IRQ4_USED_BY_ISA	6Eh	08h
Q_IRQ5_USED_BY_ISA	6Eh	10h
Q_IRQ7_USED_BY_ISA	6Eh	20h
Q_IRQ9_USED_BY_ISA	6Eh	40h
Q_IRQ10_USED_BY_ISA	6Eh	80h
Q_IRQ11_USED_BY_ISA	6Fh	01h
Q_IRQ14_USED_BY_ISA	6Fh	02h
Q_IRQ15_USED_BY_ISA	6Fh	04h

Q_MEM_SIZE_USED_BY_ISA	6Fh	18h
Q_MEM_BASE_USED_BY_ISA	6Fh	E0h
Q_IT8661F_FLOPPY_CONTROLLER	70h	03h
Q_IT8661F_COMA_PORT	70h	1Ch
Q_IT8661F_COMB_PORT	70h	E0h
Q_IT8661F_IR_PORT	71h	07h
Q_IT8661F_IR_MODE	71h	08h
Q_IT8661F_IR_IRQ	71h	F0h
Q_IT8661F_LPT_PORT	72h	07h
Q_IT8661F_LPT_MODE	72h	18h
Q_IT8661F_LPT_IRQ	72h	20h
Q_IT8661F_LPT_ECP_DMA	72h	C0h
Q_SPREAD_SPECTRUM	73h	03h
Q_FREQ_SELECT	73h	04h
Q_CPU_FREQ	73h	78h
Q_SIS5513_IDE_CONTROLLER	74h	03h
Q_SIS5513_PREFETCH_PM	73h	80h
Q_SIS5513_PREFETCH_PS	74h	04h
Q_SIS5513_PREFETCH_SM	74h	08h
Q_SIS5513_PREFETCH_SS	74h	10h
Q_SYS_HW_DISPLAY	74h	20h
Q_SYS_TEMP	74h	40h
Q_CPU_FAN_SPEED	74h	80h
Q_SYS_FAN_SPEED	75h	01h
Q_SYS_IN0	75h	02h
Q_SYS_IN2	75h	04h
Q_SYS_IN1	75h	08h
Q_SYS_IN3	75h	10h
Q_FLOPPY_A	10h	F0h
Q_FLOPPY_B	10h	0Fh
Q_COLOR	37h	0Fh
Q_VIRUS_PROTECTION	29h	20h
Q_PRI_MASTER_TYPE	12h	F0h
Q_PRI_SLAVE_TYPE	12h	0Fh
Q_SEC_MASTER_TYPE	27h	FFh

Q_SEC_SLAVE_TYPE	28h	FFh
Q_PRI_MASTER_LBA_MODE	2Ah	01h
Q_PRI_SLAVE_LBA_MODE	2Bh	01h
Q_SEC_MASTER_LBA_MODE	2Ch	01h
Q_SEC_SLAVE_LBA_MODE	2Dh	01h
Q_PRI_MASTER_BLOCK_MODE	2Ah	02h
Q_PRI_SLAVE_BLOCK_MODE	2Bh	02h
Q_SEC_MASTER_BLOCK_MODE	2Ch	02h
Q_SEC_SLAVE_BLOCK_MODE	2Dh	02h
Q_PRI_MASTER_32BIT_XFER	2Ah	04h
Q_PRI_SLAVE_32BIT_XFER	2Bh	04h
Q_SEC_MASTER_32BIT_XFER	2Ch	04h
Q_SEC_SLAVE_32BIT_XFER	2Dh	04h
Q_PRI_MASTER_PIO_MODE	2Ah	38h
Q_PRI_SLAVE_PIO_MODE	2Bh	38h
Q_SEC_MASTER_PIO_MODE	2Ch	38h
Q_SEC_SLAVE_PIO_MODE	2Dh	38h
Q_LANGUAGE	13h	1Ch

Last CMOS address used = 75h ← 最后一个被使用到的 CMOS 地址为 75h

以下省略

若是比较新版的 AMISSP.EXE (如笔者的 v2.07b)，则会非常详细地列举出每个设置屏幕群、每个提问变量、提问字符串、选项文字、选项字段，以及索引位置、默认值与开机值。

```
*********************************************************************;
*********************************************************************;
;**                                                                **;
;**          (C) Copyright 1985-1998, American Megatrends, Inc.    **;
;**                                                                **;
;**                      All Rights Reserved.                      **;
;**                                                                **;
;**            6145-F Northbelt Pkwy, Norcross, GA 30071           **;
;**                                                                **;
;**                      Phone (770) -246-8600                     **;
;**                                                                **;
*********************************************************************;
*********************************************************************;
```

AMIBIOS Setup Script Processor
Version 2.07b (AMISSP Language Version 1)
Copyright (c) 1995-1997 American Megatrends, Inc.

Input files:
c:\masm611\tmp\nme06661. (Peripheral setup items)

Screen SCR_ADVANCED ← 隶属于SCR_ADVANCED 这个 screen 族群

定义问题变量	索引值	屏蔽值	默认值	开机值
Q_QUICK_BOOT	Reg	Mask	Default	PowerOn
(Quick Boot) ← 显示的文字讯息	13h	80h	0001h	0000h

00h = (Disabled)
01h = (Enabled)

Q_PM_EMULATION_TYPE	Reg	Mask	Default	PowerOn
(Pri Master ARMD Emulated as)	52h	03h	0000h	0000h

00h = (Auto)
01h = (Floppy)
02h = (Hard Disk)

Q_PS_EMULATION_TYPE	Reg	Mask	Default	PowerOn
(Pri Slave ARMD Emulated as)	52h	0Ch	0000h	0000h

00h = (Auto)
01h = (Floppy)
02h = (Hard Disk)

Q_SM_EMULATION_TYPE	Reg	Mask	Default	PowerOn
(Sec Master ARMD Emulated as)	52h	30h	0000h	0000h

00h = (Auto)
01h = (Floppy)
02h = (Hard Disk)

Q_SS_EMULATION_TYPE	Reg	Mask	Default	PowerOn
(Sec Slave ARMD Emulated as)	52h	C0h	0000h	0000h

00h = (Auto)
01h = (Floppy)
02h = (Hard Disk)

Q_FIRST_BOOT_DRIVE	Reg	Mask	Default	PowerOn
(1st Boot Device)	53h	0Fh	0001h	0001h

External Function = BOOT_DEVICE_PROC_1

Q_SECOND_BOOT_DRIVE	Reg	Mask	Default	PowerOn
(2nd Boot Device)	53h	F0h	0005h	0005h

External Function = BOOT_DEVICE_PROC_2

Q_THIRD_BOOT_DRIVE	Reg	Mask	Default	PowerOn
(3rd Boot Device)	54h	0Fh	0000h	0000h

External Function = BOOT_DEVICE_PROC_3

Q_FOURTH_BOOT_DRIVE	Reg	Mask	Default	PowerOn
(4th Boot Device)	54h	F0h	0000h	0000h

External Function = BOOT_DEVICE_PROC_4

Q_REMAINING_BOOT_DRIVES	Reg	Mask	Default	PowerOn
(Try Other Boot Devices)	55h	01h	0001h	0001h

00h = (Yes)
01h = (No)

Q_INIT_I2O_DEVICE	Reg	Mask	Default	PowerOn
(Initialize I2O Devices)	55h	02h	0001h	0001h

00h = (Yes)
01h = (No)

Q_INDEX_I2O_BOOT_DRIVE	Reg	Mask	Default	PowerOn
(I2O Boot Device Index)	56h	FFh	0000h	0000h

Q_INITIAL_DISPLAY	Reg	Mask	Default	PowerOn
(Initial Display Mode)	55h	04h	0000h	0000h

00h = (BIOS)
01h = (Silent)

Q_ADDON_ROM_DISPLAY	Reg	Mask	Default	PowerOn
(Display Mode at Add-On ROM Init)	55h	08h	0000h	0000h

00h = (Force BIOS)
01h = (Keep Current)

Q_FDD_ACCESS_CONTROL	Reg	Mask	Default	PowerOn
(Floppy Access Control)	55h	10h	0000h	0000h

00h = (Read-Write)
01h = (Read-Only)

Q_HDD_ACCESS_CONTROL	Reg	Mask	Default	PowerOn
(Hard Disk Access Control)	55h	20h	0000h	0000h

00h = (Read-Write)
01h = (Read-Only)

Q_SMART_IDE	Reg	Mask	Default	PowerOn
(S.M.A.R.T. for Hard Disks)	55h	40h	0000h	0000h

00h = (Disabled)
01h = (Enabled)

Q_BOOTUP_SPEED	Reg	Mask	Default	PowerOn
(BootUp CPU Speed)	11h	80h	0001h	0001h

00h = (Low)
01h = (High)

Q_NUM_LOCK	Reg	Mask	Default	PowerOn
(BootUp Num-Lock)	11h	01h	0001h	0001h

00h = (Off)
01h = (On)

Q_TURBO_SWITCH	Reg	Mask	Default	PowerOn
(Turbo Switch)	13h	20h	0000h	0000h

00h = (Disabled)
01h = (Enabled)

Q_FLOPPY_SWAP	Reg	Mask	Default	PowerOn
(Floppy Drive Swap)	11h	20h	0000h	0000h

00h = (Disabled)
01h = (Enabled)

Q_FLOPPY_SEEK	Reg	Mask	Default	PowerOn
(Floppy Drive Seek)	11h	10h	0000h	0000h

00h = (Disabled)
01h = (Enabled)

Q_MOUSE_SUPPORT	Reg	Mask	Default	PowerOn
(PS/2 Mouse Support)	29h	40h	0001h	0001h

00h = (Disabled)
01h = (Enabled)

Q_TYPEMATIC_RATE	Reg	Mask	Default	PowerOn
(Typematic Rate)	11h	04h	0001h	0001h

00h = (Slow)
01h = (Fast)

Q_KEYBOARD_PRESENT	Reg	Mask	Default	PowerOn
(System Keyboard)	14h	04h	0001h	0001h

00h = (Absent)
01h = (Present)

Q_DISPLAY_TYPE	Reg	Mask	Default	PowerOn
(Primary Display)	14h	38h	0001h	0001h

00h = (Absent)
01h = (VGA/EGA)

02h = (RESERVED)
03h = (CGA40x25)
04h = (RESERVED)
05h = (CGA80x25)
06h = (RESERVED)
07h = (Mono)

Q_PASSWORD_CHECK	Reg	Mask	Default	PowerOn
(Password Check)	29h	10h	0000h	0000h

00h = (Setup)
01h = (Always)

Q_PARITY_ERROR	Reg	Mask	Default	PowerOn
(Parity Check)	29h	08h	0000h	0000h

00h = (Disabled)
01h = (Enabled)

Q_OS2_COMP_MODE	Reg	Mask	Default	PowerOn
(Boot To OS/2 > 64MB)	29h	80h	0000h	0000h

00h = (No)
01h = (Yes)

Q_WAIT_FOR_F1	Reg	Mask	Default	PowerOn
(Wait For 'F1' If Error)	11h	02h	0001h	0001h

00h = (Disabled)
01h = (Enabled)

Q_DISPLAY_HIT_DEL	Reg	Mask	Default	PowerOn
(Hit 'DEL' Message Display)	11h	08h	0001h	0001h

00h = (Disabled)

01h = (Enabled)

Q_INTERNAL_CACHE	Reg	Mask	Default	PowerOn
(Internal Cache)	57h	03h	0002h	0002h

←— 定义的子程序跳越进入点

External Function = INTERNAL_CACHE_PROC

Q_EXTERNAL_CACHE	Reg	Mask	Default	PowerOn
(External Cache)	57h	0Ch	0002h	0000h

External Function = EXTERNAL_CACHE_PROC

Q_CACHE_BIOS	Reg	Mask	Default	PowerOn
(System BIOS Cacheable)	55h	80h	0001h	0000h

00h = (Disabled)
01h = (Enabled)

Q_C0000_SHADOW	Reg	Mask	Default	PowerOn
(C000,32k Shadow)	57h	30h	0002h	0002h

00h = (Disabled)
01h = (Enabled)
02h = (Cached)
03h = (RESERVED)

Q_C8000_SHADOW	Reg	Mask	Default	PowerOn
(C800,16k Shadow)	57h	C0h	0000h	0000h

00h = (Disabled)
01h = (Enabled)
02h = (Cached)
03h = (RESERVED)

Q_CC000_SHADOW	Reg	Mask	Default	PowerOn
(CC00,16k Shadow)	58h	03h	0000h	0000h

00h = (Disabled)
01h = (Enabled)
02h = (Cached)
03h = (RESERVED)

Q_D0000_SHADOW	Reg	Mask	Default	PowerOn
(D000,16k Shadow)	58h	0Ch	0000h	0000h

00h = (Disabled)
01h = (Enabled)
02h = (Cached)
03h = (RESERVED)

Q_D4000_SHADOW	Reg	Mask	Default	PowerOn
(D400,16k Shadow)	58h	30h	0000h	0000h

00h = (Disabled)
01h = (Enabled)
02h = (Cached)
03h = (RESERVED)

Q_D8000_SHADOW	Reg	Mask	Default	PowerOn
(D800,16k Shadow)	58h	C0h	0000h	0000h

00h = (Disabled)
01h = (Enabled)
02h = (Cached)
03h = (RESERVED)

Q_DC000_SHADOW	Reg	Mask	Default	PowerOn
(DC00,16k Shadow)	59h	03h	0000h	0000h

00h = (Disabled)

01h = (Enabled)
02h = (Cached)
03h = (RESERVED)

Screen SCR_CHIPSET

Q_SHARE_MEMORY	Reg	Mask	Default	PowerOn
(Share Memory Function)	59h	04h	0001h	0001h

00h = (Disabled)
01h = (Enabled)

以下省略

(从 1456 列开始)

Q_LANGUAGE	Reg	Mask	Default	PowerOn
(Language)	13h	1Ch	0000h	0000h

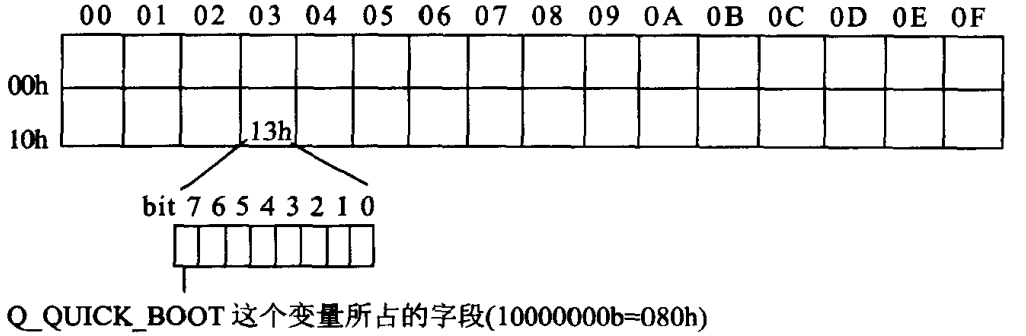
00h = (English)

Last CMOS address used = 75h ←—最后一个被使用到的 CMOS 地址为 75h

以下省略

在前面的章节已谈过，一般 CMOS 内存只有 128byte 大小，以十六进制表示其地址，就是从 00h～7Fh，换句话说，目前 BIOS 对于 CMOS 的索引位置，最大的地址上限为 07Fh，而且扣掉一些从 286/AT 时代就固定死、绝对不能更动的字段，实际上能应用来作为记录提问选项的变量存放地址，如果每个变量都要用八个位 (一个 byte) 来记录，可设计、运用的附加选项不会超过 110 个，甚至一百个都不到。

因此，AMISSP.EXE 已经竭尽所能，甚至每个变量的控制单位可以微缩到只有bit的大小。以上面的SSP.REP 为例，第一栏的 Q_QUICK_BOOT 这个变量，它的索引位置为 13h，而屏蔽值为 80h，代表它存放于 CMOS 记忆地址的第13h 处；字段则是从最高的第七位算起，占用一个位宽度，用 bit7=1 代表要使用 QuickBoot 的功能，bit7=0 代表关掉 QuickBoot。



Q_QUICK_BOOT 这个变量所占的字段(10000000b=080h)

但无论如何，在设计 BIOS 画面的选项与变量时还是要小心谨慎，不要定义过大的 Optionbits，同时在经过 MakeBIOS 之后，记得打开 SSP.REP 注意一下最后使用到的 CMOS 地址，有没有用爆掉；万一有爆掉 (大于 7eh)，则必须删减一些不必要的提问选项与变量，然后再来重新 Assembling、Linking 与 Porting，这也是目前 BIOS 工程师天天要应付的例行性作业。

7-4 直接取用源代码文件群的.SSP 来修改

事实上 AMI 在交付 BIOS 源代码时，已经附了下列一些预先作好的.SSP 文件，分别是 CORE.SSP、ADVANCED.SSP、CHIPSET.SSP、PCIPNP.SSP、POWER.SSP、IO.SSP(里面再加载了 IO1.SSP、IO2.SSP、IO3.SSP)、IOC.SSP 等.SSP 文件。下列是笔者整理所制作的交叉比较表格，读者应该可以更清楚地了解它们之间的对应关系。

BIOS 主画面选项与对映的.SSP 文件、Screen ID 之间的关系

主画面选项	对应的SSP文件	所在的目录位置	对应的 Screen ID
Standard CMOS Setup	CORE.SSP (注 1)	..\..\..\OBJ\ OEMCORE.MOD	SCR_STANDARD
Advanced CMOS Setup	ADVANCED.SSP	\..\SYS\ OEMCHIP.MOD	SCR_ADVANCED
Advanced Chipset Setup	CHIPSET.SSP	\..\SYS\ OEMCHIP.MOD	SCR_CHIPSET
Power Manage--ment SETUP	POWER.SSP	\..\GPC\ OEMCHIP.MOD	SCR_POWER
PCI/Plug and Play Setup	PCIPNP.SSP	\..\\DIM\ OEMCHIP.MOD	SCR_PCI
Peripheral Setup	IO.SSP IO1.SSP IO2.SSP IO3.SSP	..\..\..\OBJ\ ..\..\..\IO.112\ OEMCHIP.MOD	SCR_PERIPHERAL
Hardware Monitor Setup	IOC.SSP	..\..\IO\O EMCHIP.MOD	SCR_PERIPHERAL

实际上 CORE.SSP 还定义了其他项目的 QuestionID，以及最后的源代码输出叙述的定义。

至于像 Auto-Detect Hard Disks(自动侦测硬盘)、Load Optimal Settings(加载最佳化数值)、Load Fail Safe Settings(加载最保守数值)、Change Supervisor Password(更换用户密码)、Save Settings and Exit(存储设置并离开)与 Exit Without Saving (放弃存储并离开) 等选项，由于它们是属于“一选择马上作用”的单纯特性，而且被改动到的机会微乎其微，因此AMI BIOS 并没有为这些选项设计相对应的.SSP设置描述文件。

因此这些选项如果要做修改，如改为中文版或其他语言的版本，只能通过在 OEM.MAK 这个文件中，指定 SUI_FILE=otherfiles.bin(此处 otherfiles.bin 为变量，指的是文件名)，加载其他的设置界面文件。

在本书所附的光盘片中，分别有四组 SUI_FILE：HF130.BIN (HiFlex v1.30 版, 英文)、HF120C.BIN (HiFlex v1.20 版，中文)、SP132.BIN (Simple v1.32 版，英文)以及 SP120.BIN (Simple v1.20 版，英文)，可供开发者自行选择设置界面的形式。

如果有进一步的需求，可洽询 AMI 公司，或上专属网站的 FAQ 网页： http://www.ami.com/support/prodsearch.cfm?prodid=54，做进一步的洽询。

各种选项 / 参数值修改的功能限制比较表

	技术难度	可调整 / 存储参数、开机最佳值	重新指定最佳化效能值 / 最保守兼容设置值	可激活 / 隐藏某个项目	可设计、增添新的项目
用户通过 BIOS 修改	低	可以	不行 (注)	不行	不行
用户通过工具程序 (AMIBCP) 修改	中等	可以	可以	可以	不行
重新编写 .SSP语言并编译	高	可以	可以	可以	可以

最佳化效能值与最保守兼容值是预先烧写在 BIOS 内部的。

CHAPTER

8

AMI BIOS工具程序——ROM升级与编辑工具

所谓“工欲善其事，必先利其器”。为了让整个 BIOS 源代码在维护以及移植作业上的顺利，AMI 公司开发了许多有助于 BIOS 模块修改、删减、烧写，以及其他特定用途的工具程序。

本章节所要讨论的焦点，就是要依次序介绍这些工具程序的基本功能与使用语法。

8-1 AMI BIOS 基本工具程序列表

以本书所附光盘为例，工具程序均放在从根目录算起，在\627.10\EXE\底下的目录，有下列各可执行文件。

一、OEM 开发商、专业人员所使用的工具程序

工具程序文件名	功能说明	最新版本
AMIBCP.EXE	AMIBIOS Configuration Utility AMIBIOS (模块组态设置程序)	V7.00B12
AMIFLASH.EXE	FLASH EPROM Programming Utility Flash EPROM ROM烧写程序	V8.26

这里要特别说明的，就是 AMIBCP.EXE、AMIFLASH.EXE 这两个工具程序的版本更动是十分频繁的，读者(如果各位是主板负责 BIOS 的工程师)从 AMI 取得的 BIOS 源代码中，这两套工具程序的可执行文件名不见得相同。例如你手上只拿到 BCP7B12.EXE，它的意思是代表 AMIBCP.EXE，V7.00B12 的版本；FLASH826.EXE 则代表 AMIFLASH Ver8.26 版。

无论如何，这应该不会造成程序开发移植的困难，必要时更改一下文件名，rename 回来就行了。

二、建议纯 OEM 厂商研发人员所使用的工具程序

工具程序文件名	功能说明	最新版本
AMIBBP.EXE	AMI BIOS Build PreProcessor BIOS 前端处理程序，从*.BBP 生成*.mm 给AMIMM.EXE做模块增减	v2.03
BBP4MB.EXE	AMI BIOS Build PreProcessor 建立 BIOS 的前端处理程序，4MB 寻址版本	v3.01
B16CKSUM.EXE	BIOS _16_ Installation Check Structure Checksum Fixer BIOS内部模块结构检视兼检查值 修正程序	不明
CONCOPY.EXE	Console COPY Utility 控制台信息拦截的工具程序，把屏幕输出结果转成 MAKE.OUT	不明
EQUATOR.EXE	.EQU File Generation Utility 将输入的.ASM 文件另外生成一个.EQU 的前置处理 程序	v1.4
COMPRESS.EXE	将待连接模块文件压缩的工具程序	不明
PCICKSUM.EXE	PCI Installation Check Structure Checksum Fixer PCI 装置 / 模块的结构检视兼检查值修正程序	不明
PNPCKSUM.EXE	PnP Intallation Check Structure Checksum Fixer PnP模块结构检视兼检查值修正程序	不明
PLAYROM.EXE	搭配 ROMSETUP.EXE 的工具程序	不明
RP.EXE	文件特定字符串搜寻、替换的工具程序	v1.03

AMIMM.EXE、AMIDSP.EXE、AMISSP.EXE、ASL.EXE 也是搭配 MAKEBIOS.BAT、MAKEALL.BAT 编译搭配的重要工具程序

三、搭配 MAKEBIOS.BAT、MAKEALL.BAT 作编译、移植用的工具程序 (建议不要单独执行)

工具程序文件名	功能说明	最新版本
AMIBBP.EXE	AMI BIOS Build PreProcessor BIOS 前端处理程序，从 *.BBP 生成 *.mm 给 AMIMM.EXE 做模块增减	v2.03
BBP4MB.EXE	AMI BIOS Build PreProcessor 建立 BIOS 的前端处理程序，4MB 寻址版本	v3.01
B16CKSUM.EXE	BIOS _16_ Installation Check Structure Checksum Fixer BIOS 内部模块结构检视兼检查值修正程序	不明
CONCOPY.EXE	Console COPY Utility 控制台信息拦截之工具程序，把屏幕输出结果转成 MAKE.OUT	不明
EQUATOR.EXE	.EQU File Generation Utility 将输入的.ASM 文件另外生成一个.EQU 的前置处理程序	v1.4
COMPRESS.EXE	将待连接模块文件压缩的工具程序	不明
PCICKSUM.EXE	PCI Installation Check Structure Checksum Fixer PCI 装置 / 模块的结构检视兼检查值修正程序	不明
PNPCKSUM.EXE	PnP Intallation Check Structure Checksum Fixer PnP 模块结构检视兼检查值修正程序	不明
PLAYROM.EXE	搭配ROMSETUP.EXE 的工具程序	不明
RP.EXE	文件特定字符串搜寻、替换的工具程序	v1.03

至于链接 / 维护 BIOS 所需要的批处理文件，就属于 **MAKEALL.BAT**，以及**MAKEBIOS.BAT**这两个批处理文件。

四、旧版的工具程序

以下这些工具程序是属于7/15/1995之前旧结构所开发／应用的工具程序，读者(或系统开发者)也许会在AMIBIOS的技术文件中看到，或者从AMI取得的源代码看到这些可执行文件，由于这些可执行文件已经不适合于v6.30(以及本书介绍的v6.27)的BIOS结构，无法在新的BIOS影像文件使用。

AMIBCPE.EXE	AMIBIOS Configuration，针对626之前的版本，目前新版本BIOS已改由AMIBCP (V7.00)之后的工具程序所取代。
ROMINFO.EXE	旧版BIOS影像文件的模块信息列举程序，仅适用于旧版本，现已经淘汰。

五、需要读者自行准备的工具程序文件

以下这些文件，除了DEBUG.EXE (或旧版DOS的DEBUG.COM)工具程序，是DOS或Windows本身有附的之外，其他如NMAKE.EXE、NMAKER.EXE (Make Utility)、ML.EXE、MASM.EXE (Microsoft Macro Assembler V6.11)，以及LINK.EXE等可执行文件，是附在MASM611的软件开发套件上的，一般来说，只要是微软的程序发展工具，在它们的光盘片中一定找得到这些文件，这部分就请读者自行设法取得了。

最后，在本章节的规划中，我们将以第一、第二类工具程序作逐一介绍，至于第三类工具程序，由于它属于AMI BIOS内部编译与链接时的辅助工具，仅在前面做简要的说明即可，而第四项工具程序已经不适用v627.10的BIOS结构，自然我们也不需要再介绍了。

8-2 先从烧写 BIOS 开始——强大的 AMIFLASH ROM 烧写程序

AMIFlash是AMI 公司所开发的专门为 Flash EPROM 内存烧写 BIOS 监控程序的工具程序。

笔者之所以认为它强大的理由有下列几点。

① AMIFlash 除了可以在纯 DOS 模式 (或者说真实模式) 环境下操作，也可以在 Windows 环境下执行。

以往 BIOS 厂商所提供的 Flash 烧写程序，都被限制为只能在 DOS 环境下才能执行，这是因为 Flash 烧写程序需要直接动用硬件 IO 寄存器，去做提升电压、写入的细微动作，这种动作只能在 Real Mode (真实寻址模式)下工作，不可以进入 Windows、OS/2、Linux 下执行，甚至连 DOS 开机文件中的 CONFIG.SYS，都不可以指定加载 EMM386.EXE、QEMM386.SYS 之类的内存控制软件，因为它们会切换到虚拟 86 模式 (Virtual 8086mode)以及保护模式 (Protected Mode)，这些模式对应用程序直接驱动 I/O 通常都会有所限制(除非在 Ring0 环境下，才能直接动到 I/O 端口) 。

当然要在传统 DOS 的环境下写些小巧的应用程序是比较容易的，而且在真实模式下使用任何硬件 I/O 资源完全不受限制，但也相对的被 DOS 环境所限制住，要执行这些工具程序时，代价就是还得先找片 DOS 磁盘开机，或者先将 BIOS 影像文件拷贝到硬盘后，再切到 DOS 模式下开机，然后乖乖地执行 DOS 命令。

AMIFlash 程序的主结构，仍然是以 DOS 下的环境用 C 语言所开发，但是它这套C语言内嵌了 DOS/4G 的 DOS Extender(DOS 延伸系统)，可以切换到保护模式下，享有最大的内存与 I/O 的掌控能力，最重要的是它还遵循着 VCPI (Virtual Control Program Interface)与 DPMI (DOS Protected Mode Interface，DOS 保护模式界面)，能够与 Windows 主核心兼容并存，因此无论是在纯 DOS 环境，还是挂了 EMM386 的 DOS 环境，或者在 Windows 环境下执行，都可以做 Flash EPROM 的烧写动作。

② 具备双使用模式：初学者容易上手的菜单 / 控制光条界面，以及可搭配批处理文件的传统 DOS 命令参数列的方式使用。

对初学者适用的菜单 / 控制光条界面的操作方式，稍后再作介绍。我们先介绍一下 AMIFlash 在批处理模式下，后面可以接的参数语法与用途。

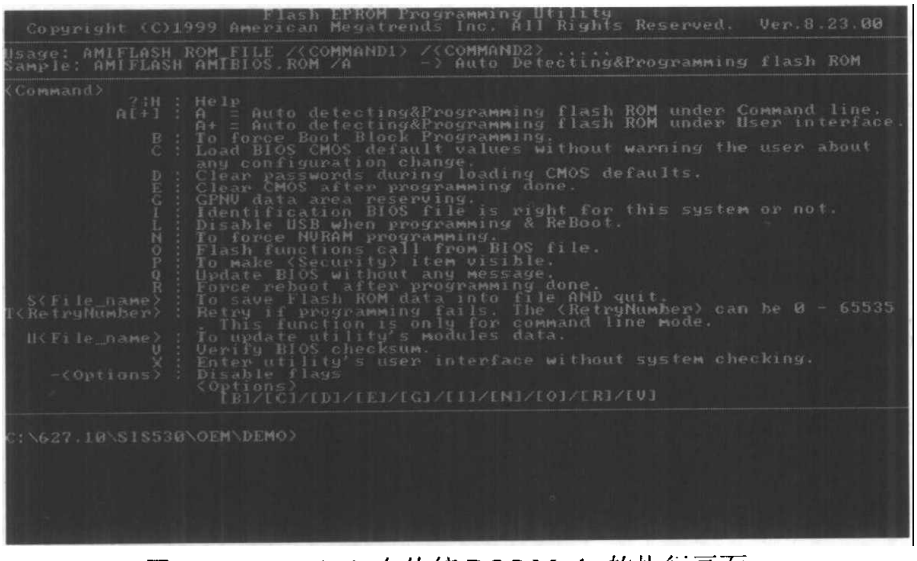


图 8-1 AMIFlash 在传统 DOS Mode 的执行画面

? 或 H：显示求助信息。例如输入 AMIFLASH?enter 或 AMIFLASH H enter，就会列出所有的附加参数与语法。

A：自动侦测 Flash EPROM 种类并执行烧写，以 DOS 命令模式下配合批处理文件。

A+：自动侦测 Flash 内存种类并执行烧写（以图形使用者界面模式）。

B：To force Boot Block Programming。连 Boot Block 区段也一并烧写。

C：加载 BIOS 的 CMOS 默认值而不提醒使用者（即使有任何组态改变）。

D：加载 CMOS 默认值时先清除密码。当新版 BIOS 的功能、提问选项有重大变动时，最需要加此参数。

E：烧写完成后顺便清除 CMOS 内存残留的参数值。

G：保留 GPNV(General Purpose Non-Volatilize)内存区域。

I：Identification BIOS file is right for this system or not。辨识目前这块 BIOS 是否属于目前这个系统的规格(主板平台)，或者与它高度兼容。

L：Disable USB when programming & ReBoot。烧写与重新开机的过程中先暂时关掉 USB 界面接口。

N：To force NVRAM programming。强迫写入 NVRAM 。

O：通过 BIOS 功能调用来进行快速烧写的动作。这是在 AMIFLASH 无法侦测到你手边的 FlashEPROM 内存时，可以加此参数来试图进行烧写。

P：把所有隐蔽的项目完全打开。

Q：更新 BIOS 但不显示任何信息。

R：ROM 烧写完毕后强制 reset 系统重新开机。

/-z：针对 8.27 版本之后的 AMIFlash.exe，用这个参数指定进入菜单 / 窗口操作界面模式 。

S<File_name>：将 FlashROM 数据存为 file_name 文件并结束执行。

T<RetryNumber>：设置烧写失败后的重试次数，从 0～65535。本参数仅限 DOS 命令模式。

U<File_name>：更新 AMIFLASH 内部的芯片参数定义文件。

V：检查 BIOS 的检查码确保数据正确性、完整性。

X：不对系统做检测，直接进入使用者界面模式。

-<选项>：关掉上述的功能标志。例如 -O、-P，就是关闭“通过 BIOS 来烧写”，以及取消“所有隐密选项开启功能”。

AMIFLASH 的使用者界面操作模式 BIOS 烧写

而所谓的使用者界面 (User Interface) 模式，就是指在 DOS 提示符号下，执行 AMIFLASH.EXE(或 FLASH826.EXE)时，烧写程序会出现如图8-2所示的画面。

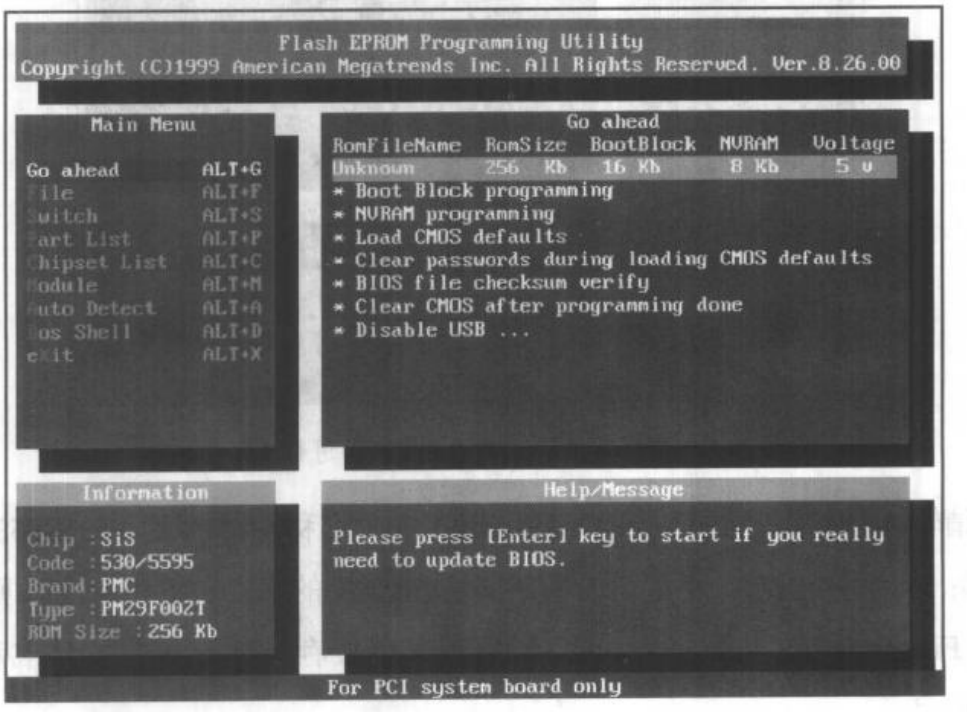


图 8-2　执行 AMIFLASH.EXE 所出现的烧写程序画面

此时一切操作方式是以[redacted]、[redacted]、[redacted]以及热键组合的方式来移动控制光条以及选择项目，这就是所谓的使用者界面模式。

最新的 8.27 版，则默认为 DOS 命令模式，读者可以用 /-z 这个参数，指定进入菜单 / 窗口操作界面模式。

读者可以发现图8-3中，由于还没有指定要加载哪个 BIOS 文件进行烧写，所以在右上方 RomFileName 字段出现 Unknown 字样，此时可以将光条下移到 File 的主选项字段后按下 [redacted](或按下 [redacted] + [redacted] 热键组合)，画面右侧会出现一个上下栏的对话窗口。

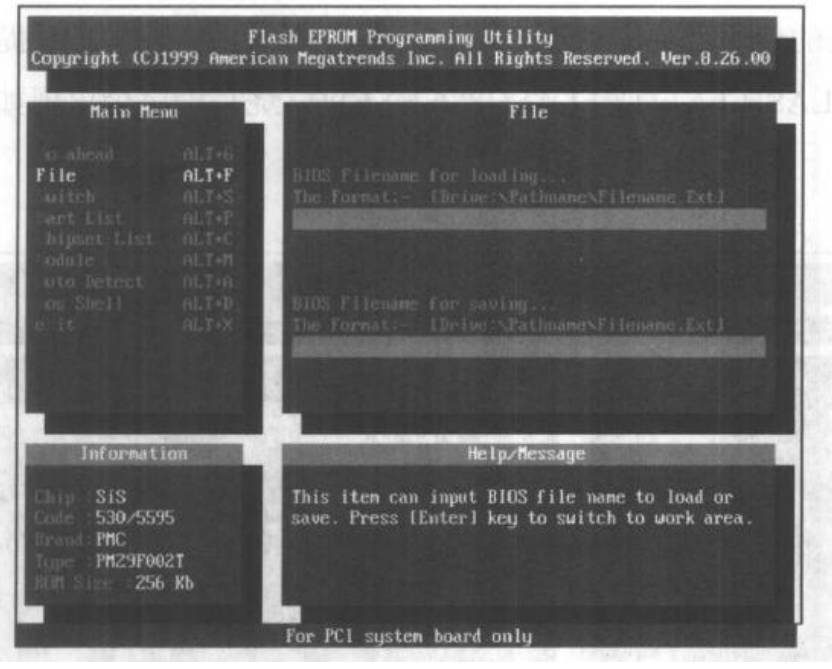


图 8-3　没指定加载哪个 BIOS 文件进行烧写的情形

请输入你要烧写到FlashROM 的BIOS 文件名称(此处我们以1SIS5300.ROM 为例)；如果你也想顺便把目前存在 FlashROM 的 BIOS 备份起来，则切换到 BIOS Filename for saving 的字段，然后输入你要存放的路径与文件名称，如图 8-5 所示。

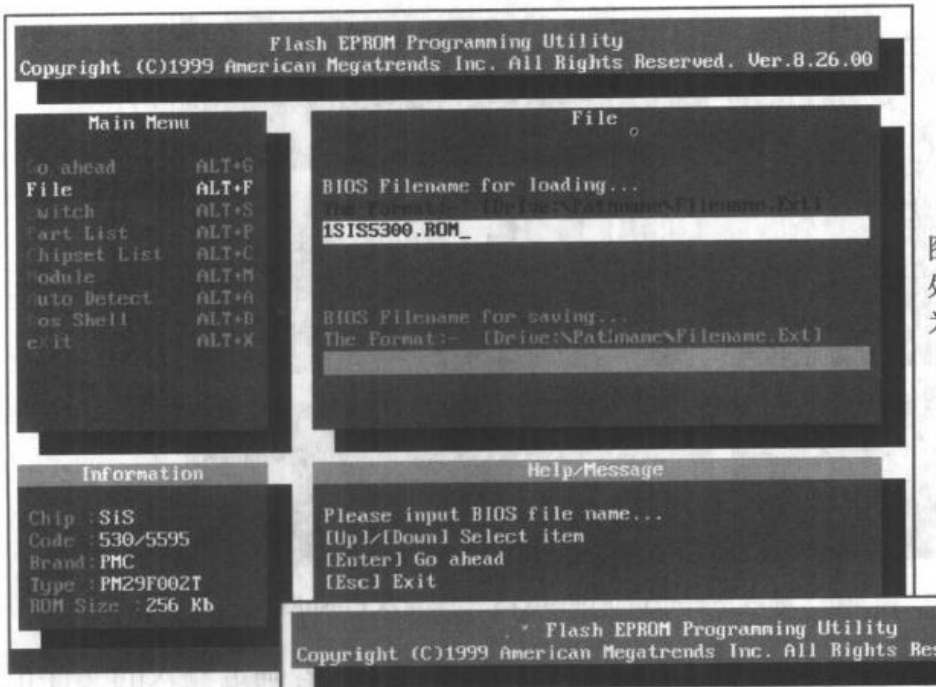


图 8-4 请加载文件(此处我们以 1SIS5300.ROM 为例)

图 8-5 把原来的 BIOS 备份起来

按如图8-4所示输入要写入FlashROM的BIOS文件名后按下 [redacted]，会弹出如图 8-6 的窗口，问你是否就是把这个指定的 BIOS 文件写进 FlashROM，此处我们不要马上按 [redacted] 键，因为还有可能有其他选项没设置好，此处先按 [redacted] 跳开，然后参阅下一段“AMIFLASH 的其他选项设置”。

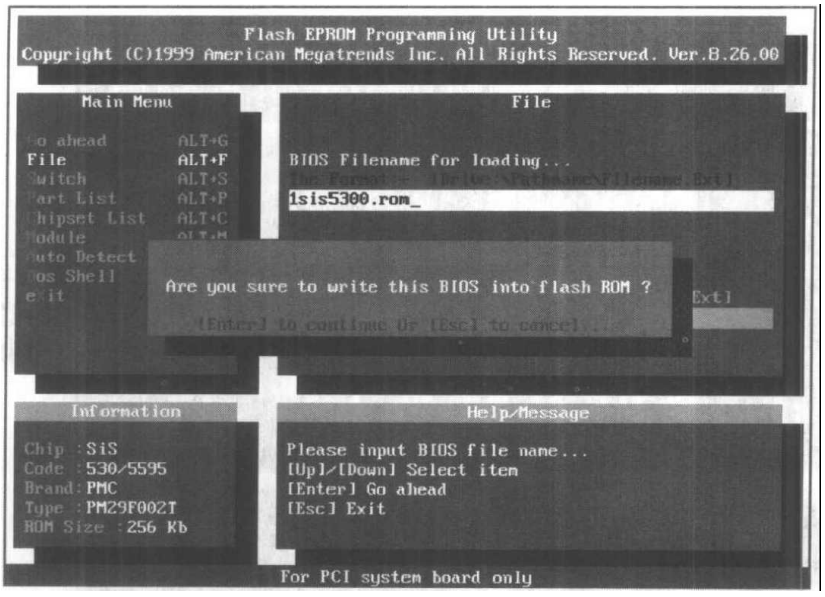


图 8-6　确认是否将指定的 BIOS 文件写入 FlashROM

当然你要是之前确定AMIFLASH已经设置妥当，或者确定输入的文件正确，当然就可以直接按下 [redacted]，进行烧写作业了。

读者也可以直接接一个刚 Porting 好(或刚下载)的 BIOS ROM影像文件，此处我们仍以 1SIS5300.ROM 为例。在 DOS 下输入 AMIFLASH 1SIS5300.ROM 时，则会出现如图 8-7 所示的画面。

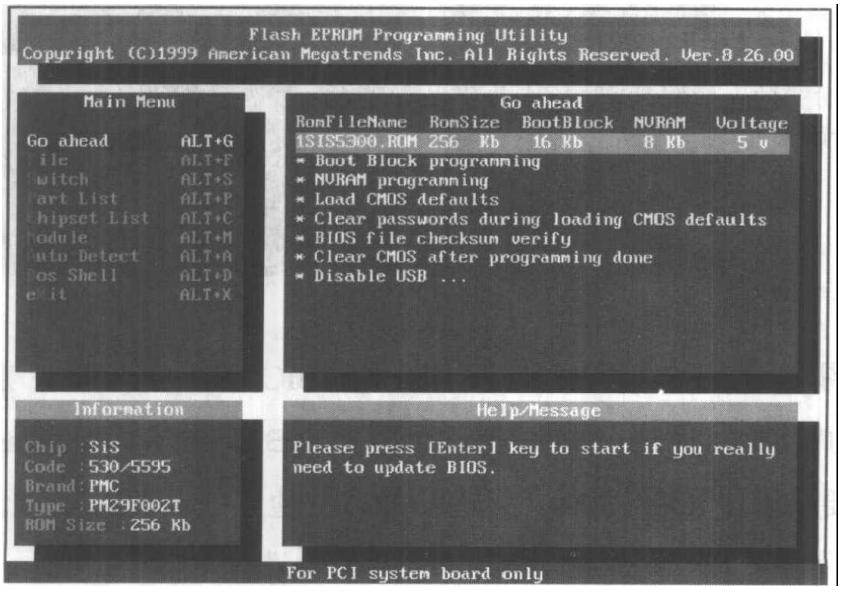


图 8-7　在 DOS 下输入 AMIFlASH 1SIS5300.ROM 时的画面

此时由于控制光条列在第一项 Go ahead(进行烧写)选项，所以按下 Enter 后，会出现如图8-8所示的提示窗口，再按下 Enter 就开始进行烧写了，如图8-9所示。

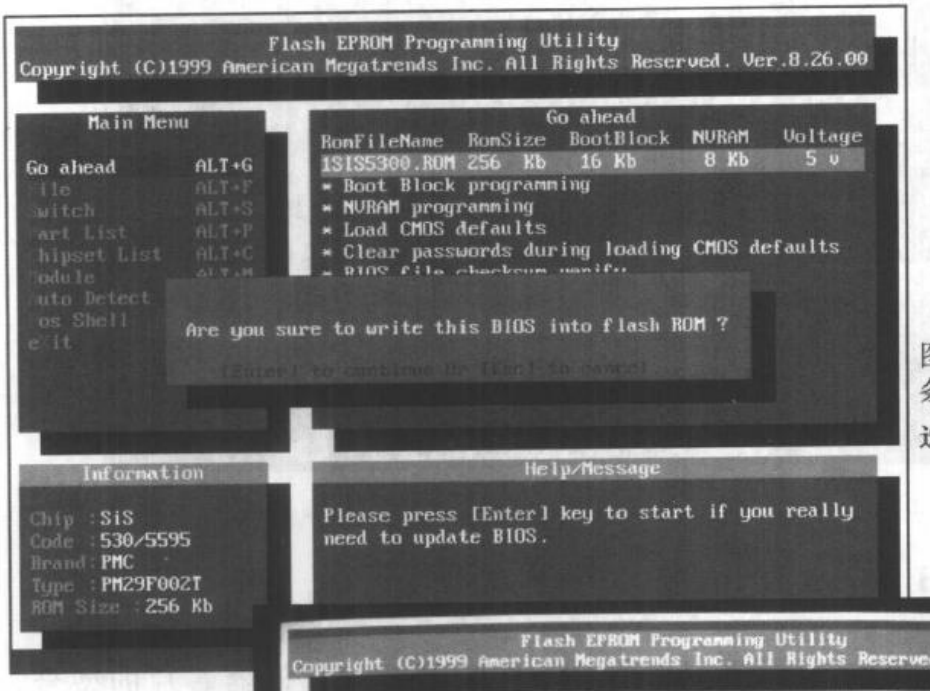


图 8-8 将控制光条选在 Go ahead 选项并按下 Enter

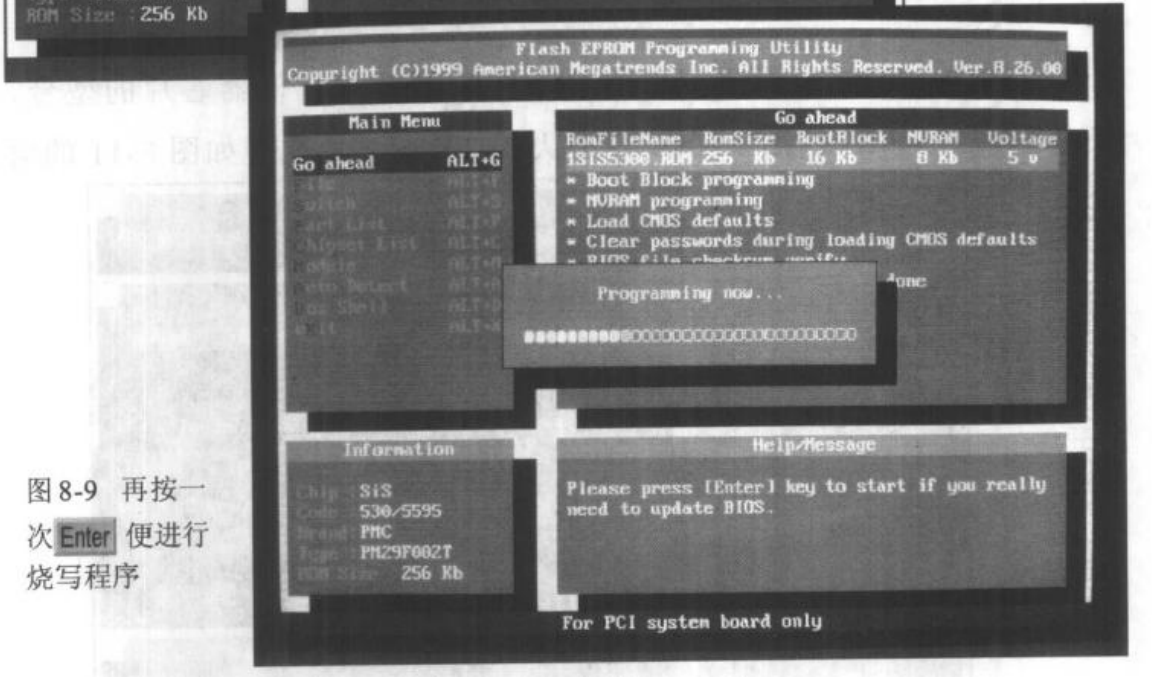


图 8-9 再按一次 Enter 便进行烧写程序

写入完成后，AMIFLASH 会出现如图8-10所示的写入完成信息，此时按任何键回到主菜单；或者等待几秒钟之后，AMIFLASH.EXE 会自动复位(Reset)开机。

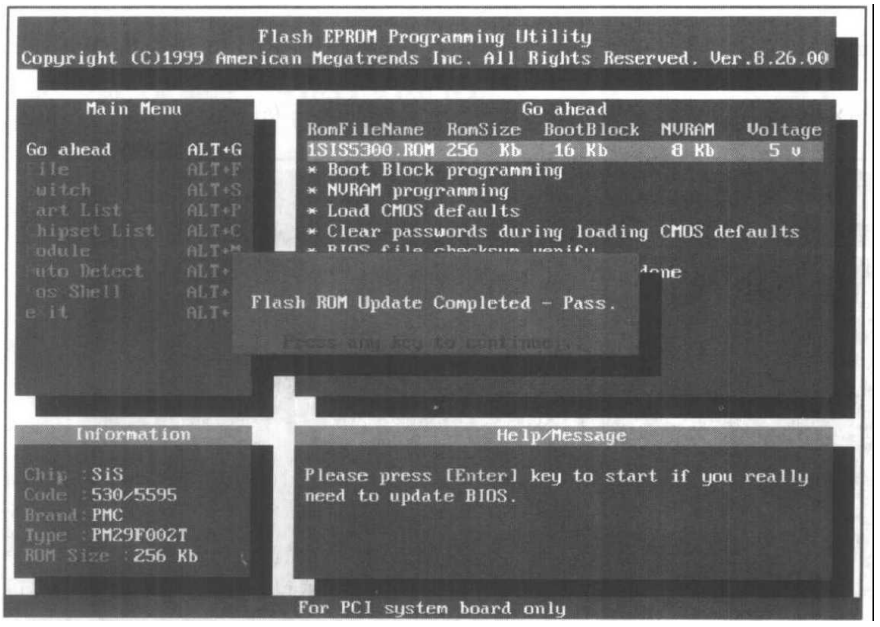


图 8-10　FLASH ROM 已写入完成

AMIFLASH 的其他选项设置

有时候，还可能因为读者主板上的快速存取存储器芯片的型号、种类，当前的 AMIFLASH.EXE 并不认得，此时它会出现如图 8-11 的信息。

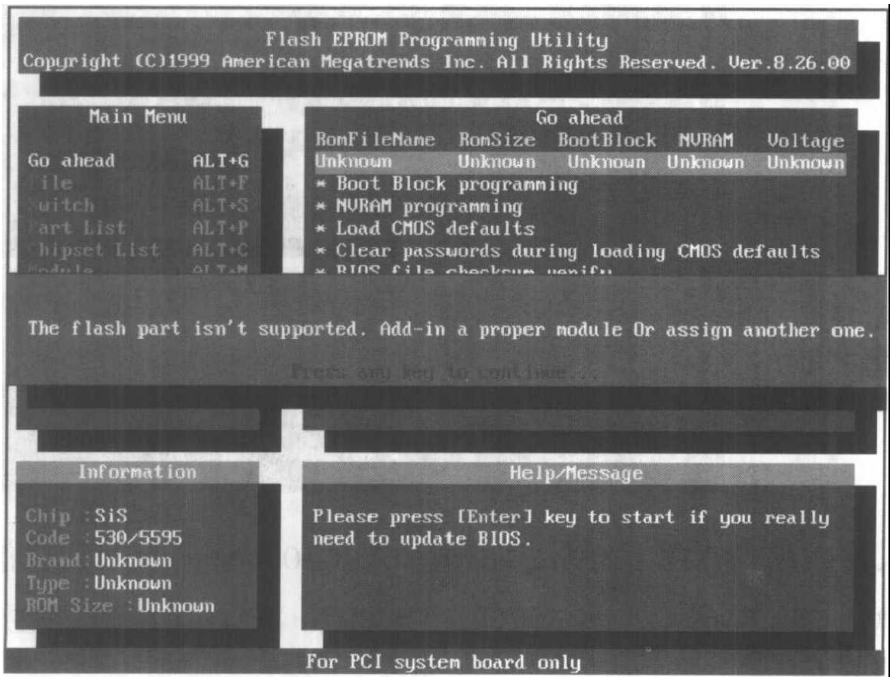


图 8-11　出现不支持某些芯片型号、种类的信息

此时你可以按手中快速存储器零组件的品牌与型号来作设置。首先如图 8-12 所示，将控制光条移到 Part List 字段并按[redacted]键(或直接按 [redacted]+[redacted] 键)，此时会切到图 8-13 所示的快速存储器零组件(Flash Part)选择窗口。

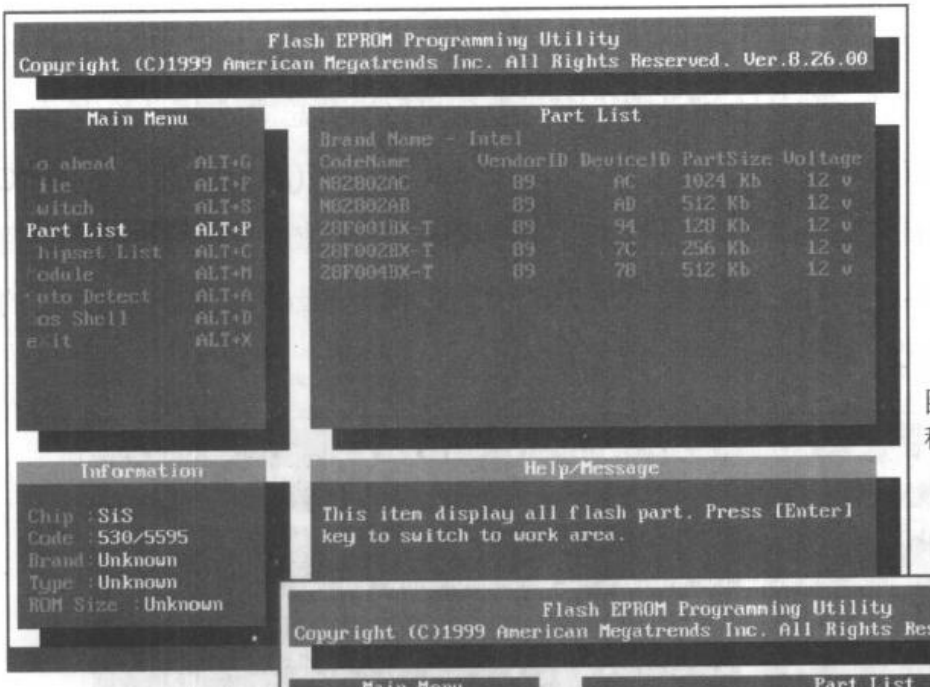


图 8-12 将控制光条移到 Part List 字段

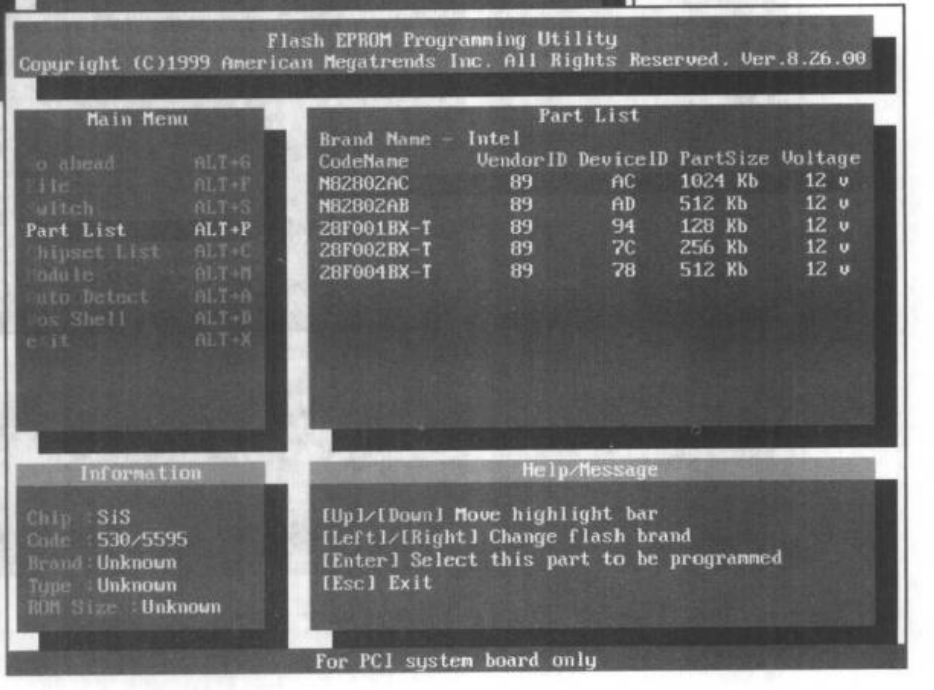


图 8-13 切入快速存储器零组件选择窗口

目前AMIFLASH.EXE V8.26版本，支持Intel、AMD、Fujitsu、ATMEL、SGS-Thomson、WinBond(华邦)、SST、IMT、Mosel Vitelic(茂硅)、MXIC(旺宏)、CATALYST、Bright、PMC与EON等共14家品牌的128KB、256KB与512KB FlashROM内存。你可以按[redacted]、[redacted]键来选择品牌，用[redacted]、[redacted]选择该品牌的某一型号(通常是以容量/烧写电压来区分)，总之选择一个与你手边Flash Part相同(或相近)的厂商/型号。此处我们以旺宏(MXIC)28F2000TPC(256KB、12V)为例(如图8-14所示)，选好种类按下[redacted]键，此时会出现如图8-15所示的画面，我们注意到左下角Brand、Type、Size原理Unknown的状态已经有了改变。

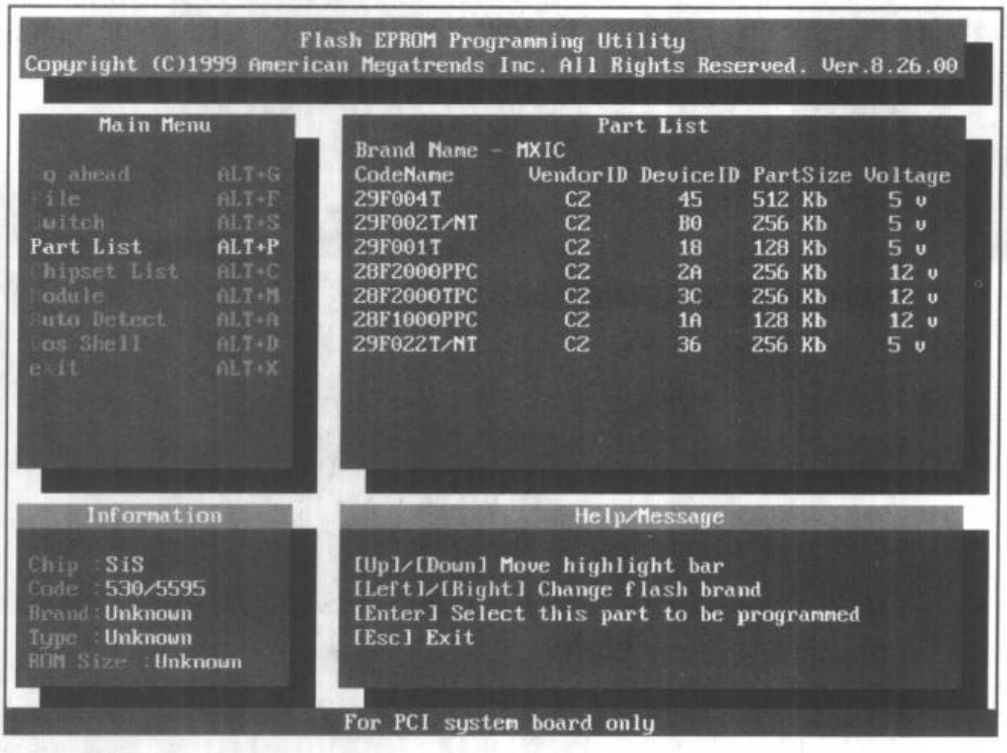


图8-14 此处以旺宏28F2000TPC为例

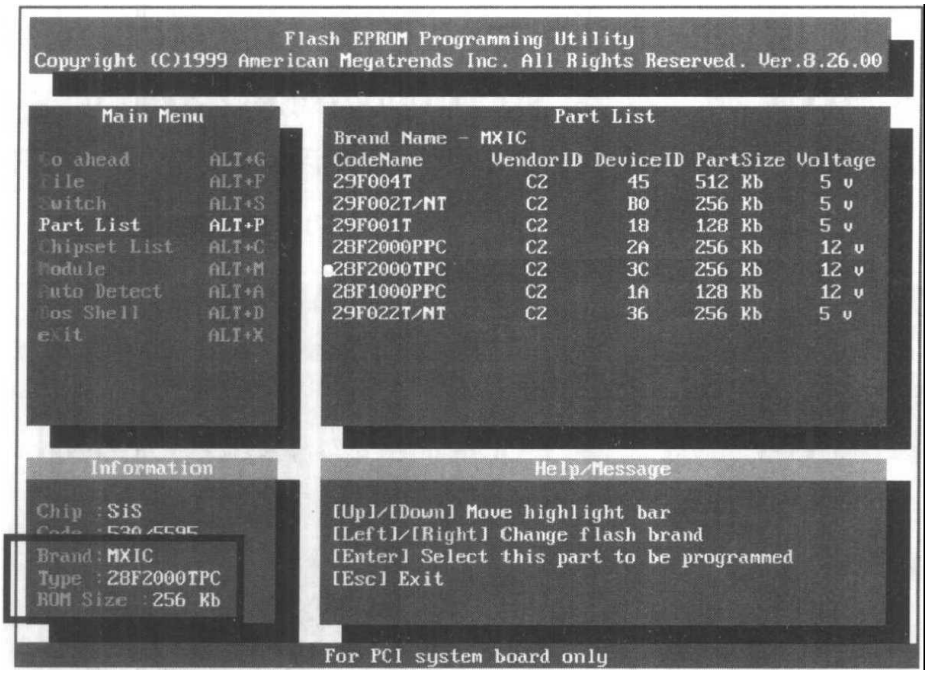


图 8-15 左下角 Brand、Type、Size 已有变化

同样，若你发现左下方主板芯片组侦测型号不正确(虽然这机会很微小)，或者有其他特定用途，例如以救援方式，要烧写不同芯片组／品牌的主板BIOS，要选择其他系统芯片组时，这个芯片组强制切换的功能就非常好用。

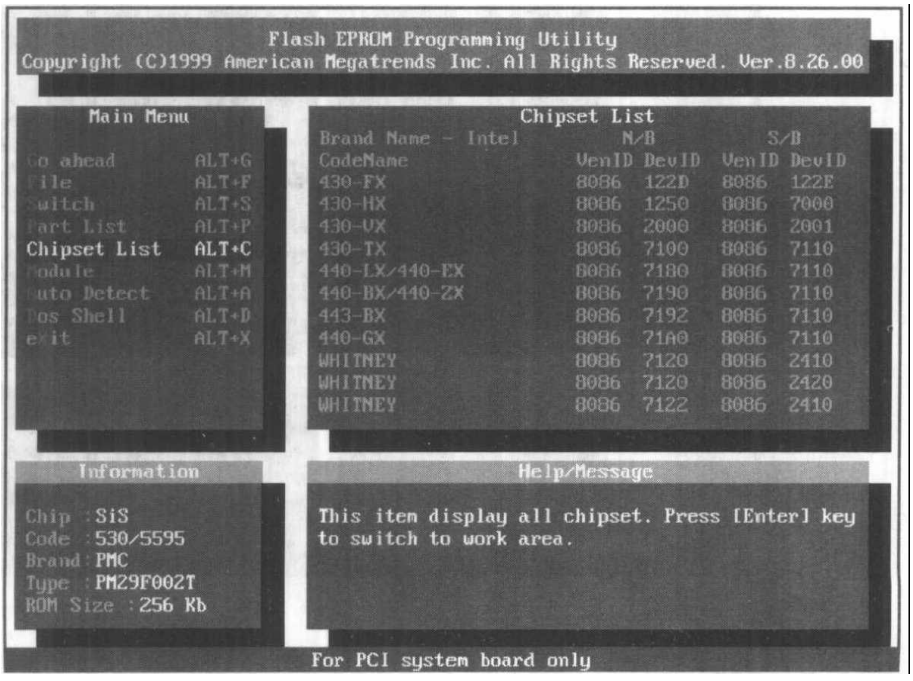


图 8-16 实用的芯片组强制切换功能

此处我们以 Intel Solano 芯片组为例，先切换到Chip List 字段，如图 8-17 所示按[redacted]键，然后用[redacted]、[redacted]键切换芯片组品牌名称，用[redacted]、[redacted]键切换到你所要选择的芯片组型号如图8-18所示，然后按下[redacted]，此时左下方的芯片组品牌 / 型号就如您所愿地指定好了，如图 8-19 所示。

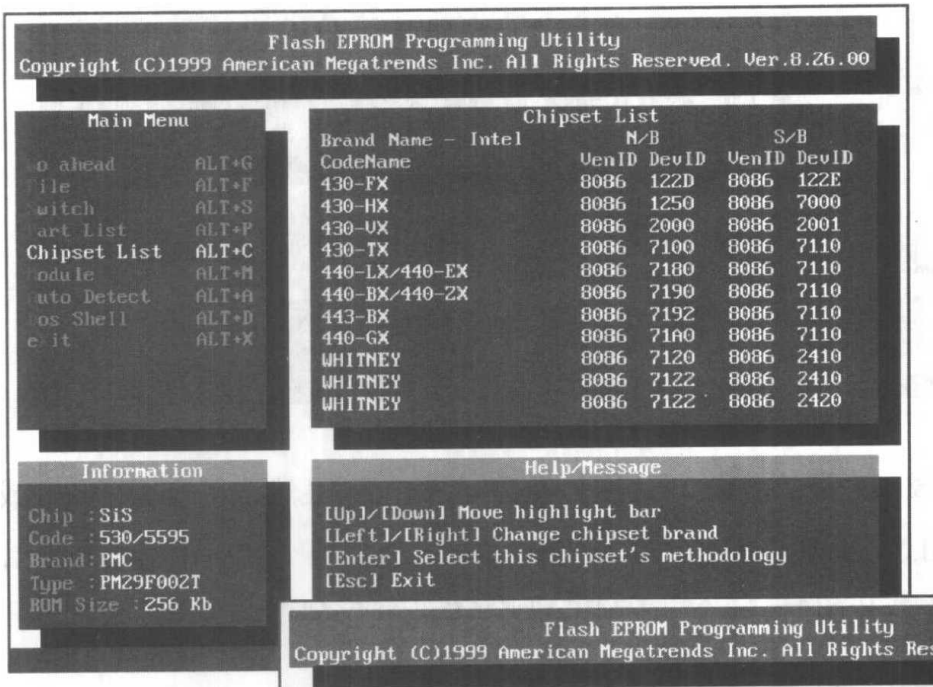


图 8-17　切换到 Chip List 字段

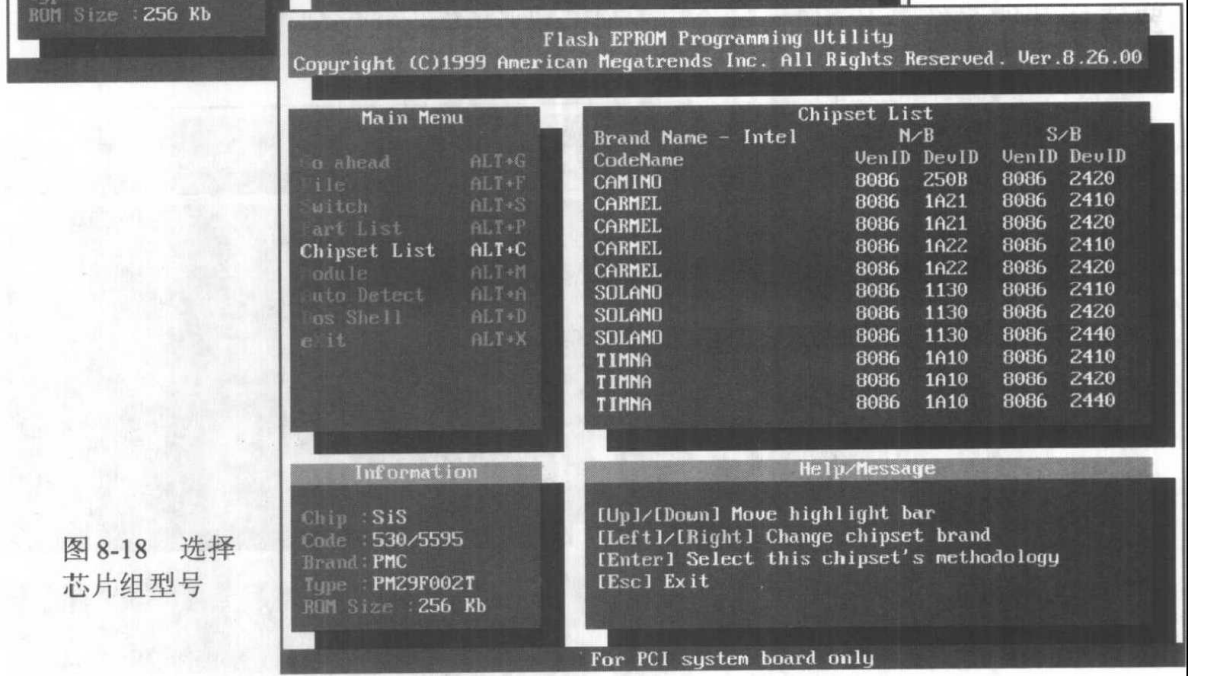


图 8-18　选择芯片组型号

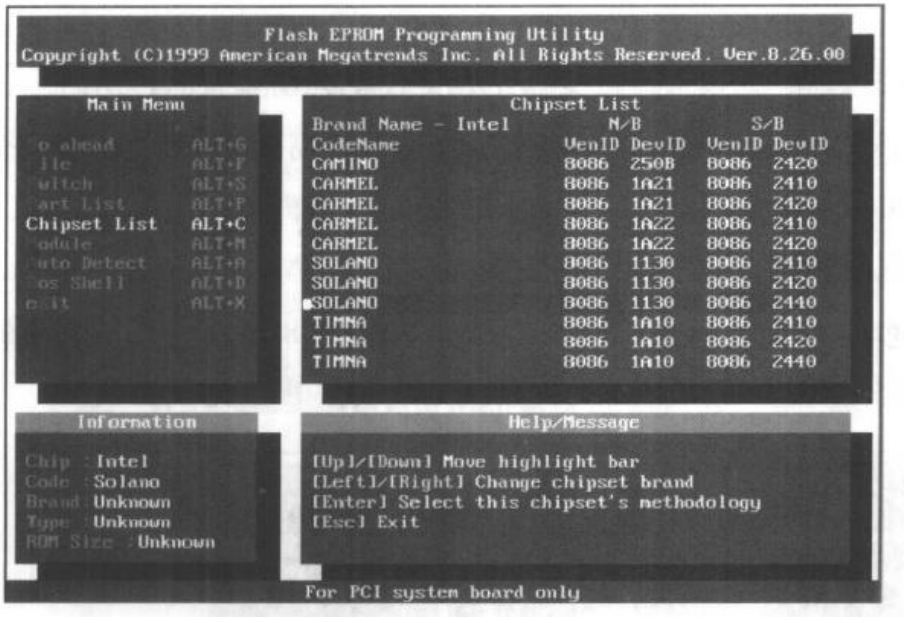


图 8-19　指定芯片组

AMIFLASH 目前支持 Intel、AMD、VIA、SIS、ALI 五大芯片组，以及多达数十种的各式芯片组，应该足够读者 / 系统研发者自由运用了。

上面介绍了 Go Ahead、File、Part List 等项目，接下来我们依序介绍其他的主选项。首先介绍 Switch(选项开关)，控制光条移到 Switch 后出现如图 8-20 所示画面。

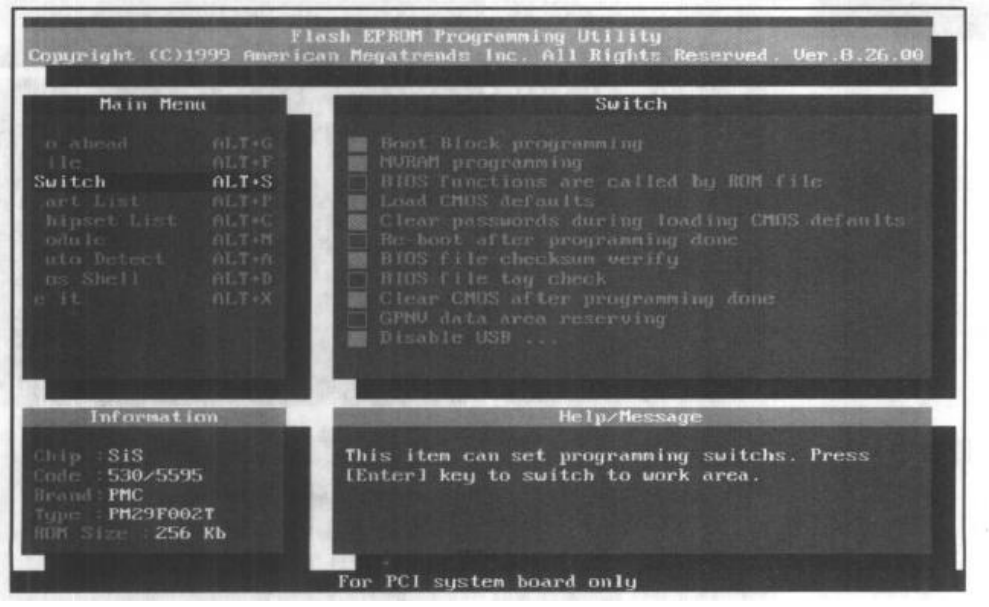


图 8-20　移动控制光条到 Switch

此时按Enter或按下Alt+S组合键，就会切到右侧，这时候可用↑、↓键选择字段，被选到的选项字段文字会呈现的高亮的状态。至于每个字段的定义，其实跟前面介绍过的在DOS批处理文件作业使用的/B、/C、/D、...、/V的参数意义相同。此处我们想设置写入完成后立刻Reset，按↓到Re-boot after programming done的字段呈现高亮时(如图8-21所示)，按下→键来选择开启(如图8-22所示)，若日后再移到该字段按Enter则切回关闭。

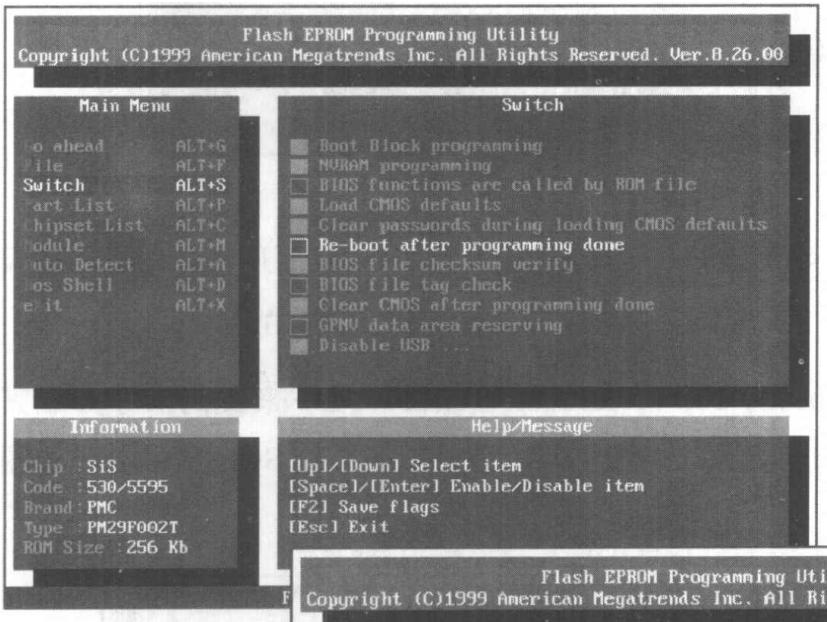


图8-21 高亮状态的字段

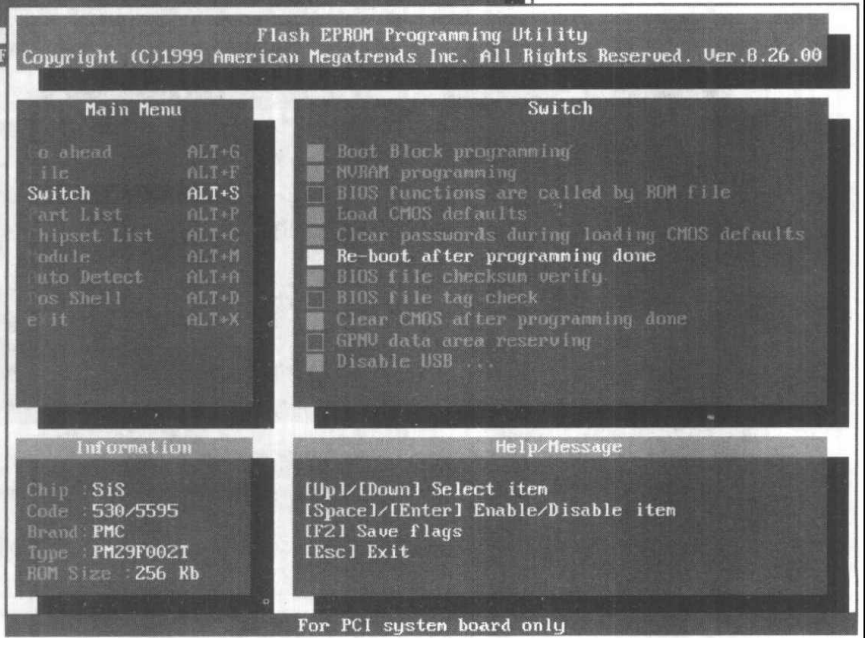


图8-22 按Enter开启

一切设好后按下[redacted]键，AMIFLASH.EXE 会如图8-23所示提醒是否确定将改过的标志位状态保存到AMIFLASH.EXE(当你切换离开AMIFLASH时忘了把Switch 选项状态保存时，也会出现此信息)。

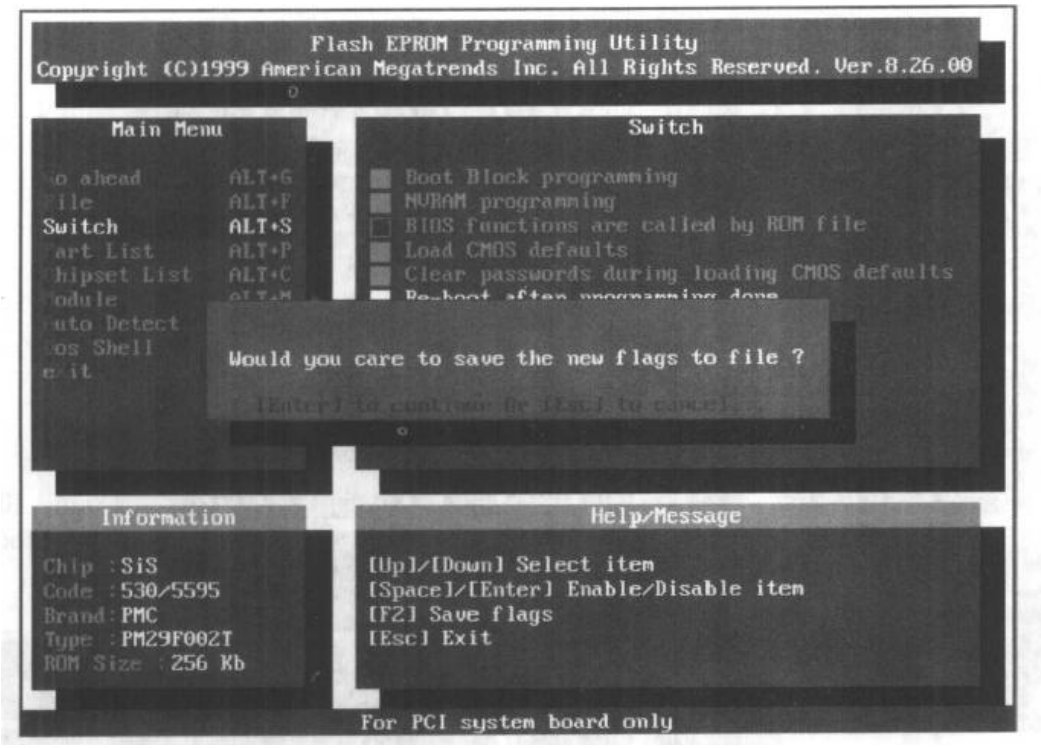


图 8-23　确认是否将改过的标志位保存

此时按[redacted]就保存设置好的选项开关状态。

笔者所建议要开启的选项开关为：NVRAM Programming、Load CMOS Default、Clear password during loading CMOS defaults、BIOS file checksum verify、Clear CMOS after programming done，以及 Disable USB。如果写入的程序代码长度不足以塞入FlashROM芯片时，除了设法删减掉多余的模块 / 子程序之外，把Boot Block Programming(连激活区块也写入)的功能打开也是一项应急的做法。

最后，我们要介绍AMIFLASH.EXE最具弹性的模块扩充型功能。当您选择 Module 字段(如图 8-24)，配合[redacted]、[redacted]、[redacted]、[redacted]、[redacted]、[redacted]等按键，来切换 AMIFLASH.EXE 内部所支持的任何芯片组模块、快速存储器种类模块(如图 8-25 所示)。

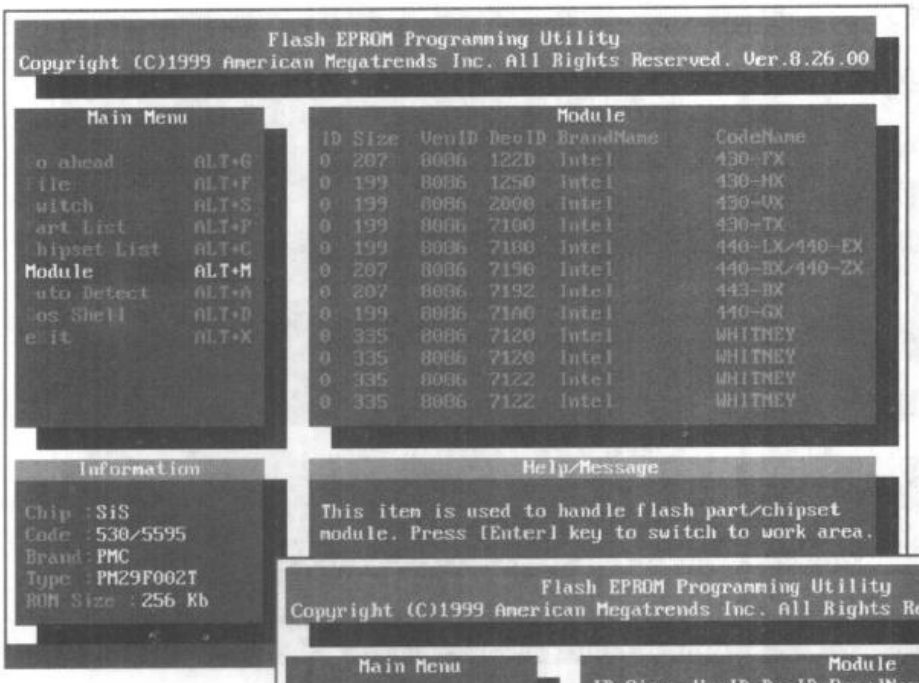


图 8-24　将控制光条移到 Module 字段

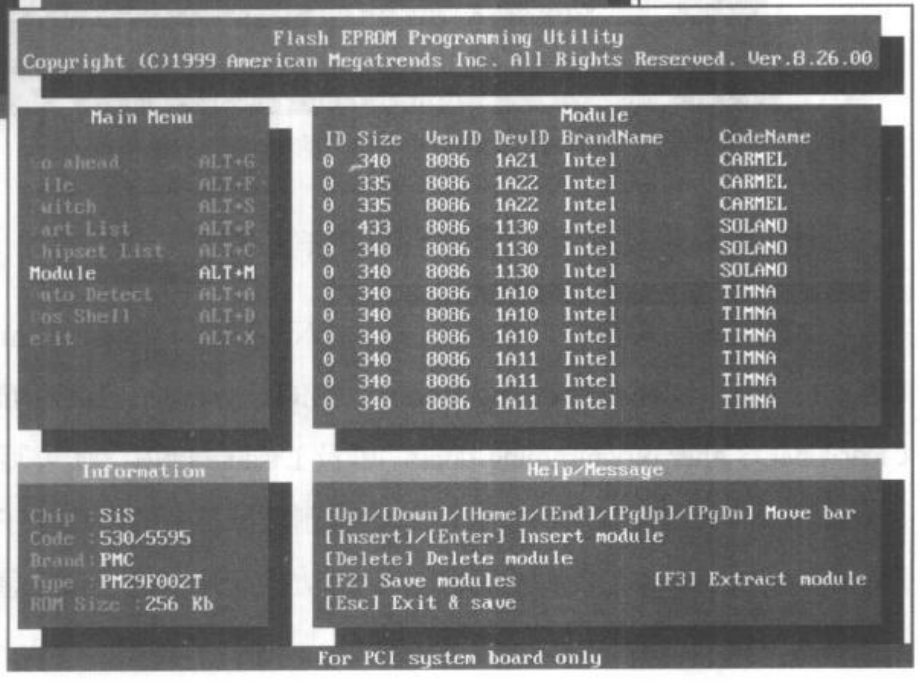


图 8-25　显示所支持的任何芯片组模块、快速存储器种类

为了示范 AMIFLASH 的模块删减、添加功能，我们先以那个早已被淘汰多年的430FX 这个模块为例(万一操作失败，顶多丢掉这个几乎用不到的模块，如图 8-26)。首先我们把字段选在 430FX 这里，按下[redacted]键把这个 430FX 芯片参数模块解压取出，并命名为 430FX.FMF(如图 8-27 所示)。

图 8-26 输入 430FX 模块并命名为 430FX.FMF

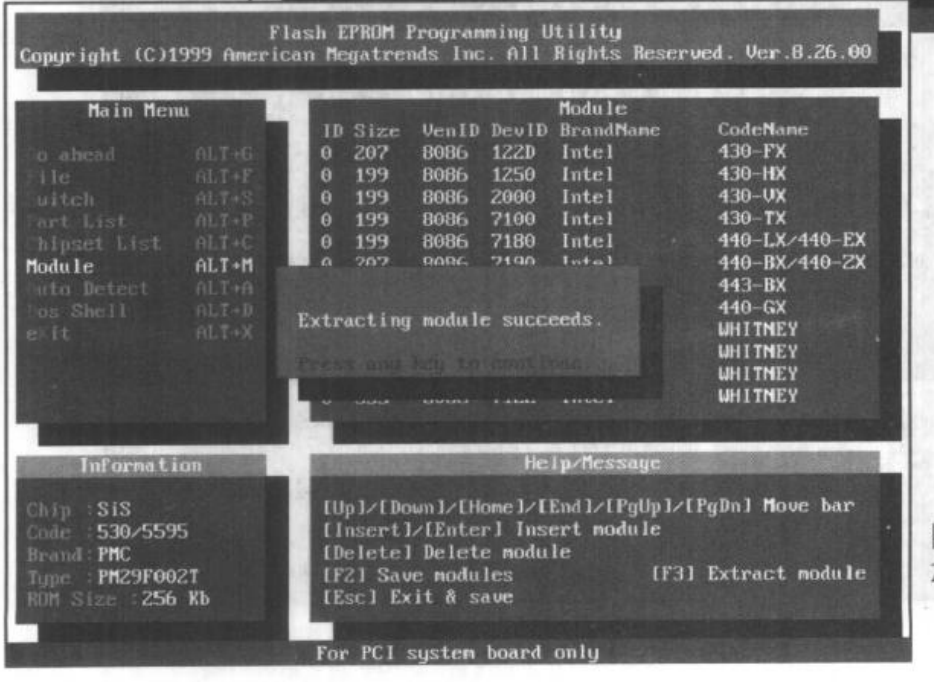


图 8-27 将 430FX 芯片参数解压缩

解压取出来后，接下来如故意按 键，把430FX这个芯片参数模块delete掉，如图 8-28、图 8-29 所示。

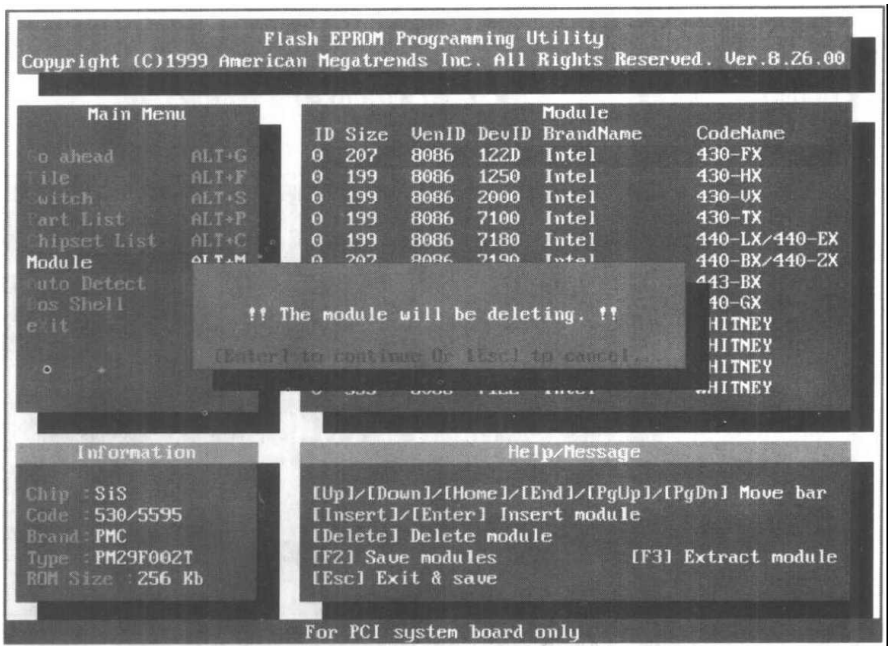


图 8-28　430FX 此芯片参数模块将会被删除

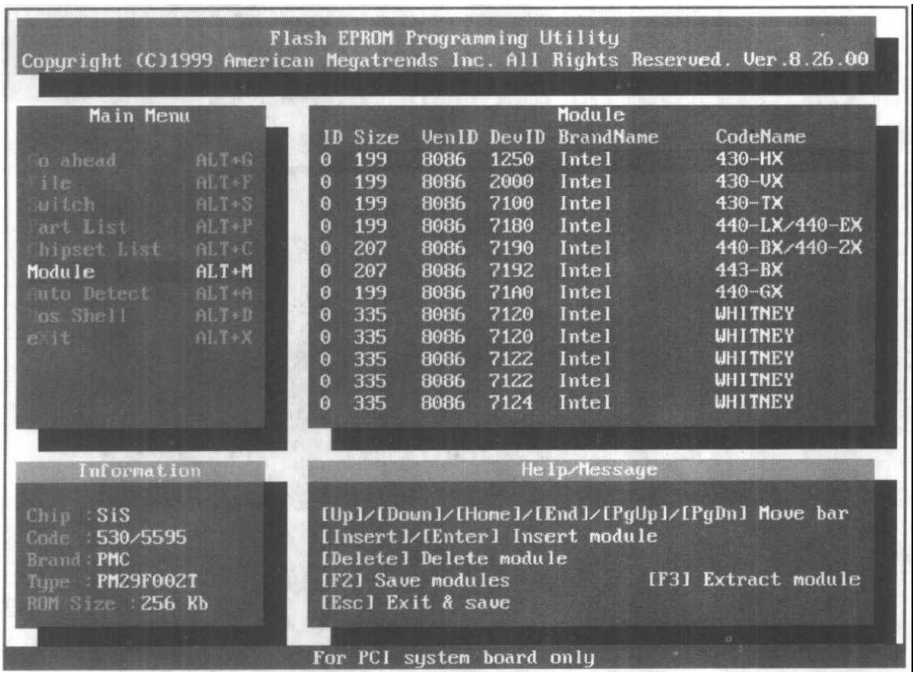


图 8-29　430FX 此芯片参数已被删除

此时按[redacted]键后按[redacted]，把删除430FX 的模块数据库保存 AMIFLASH.EXE 本身(如图8-30)。当我们再一次加载 AMIFLASH.EXE 时，你会发现430FX 这个芯片数据真的不见了(如图8-31所示)。

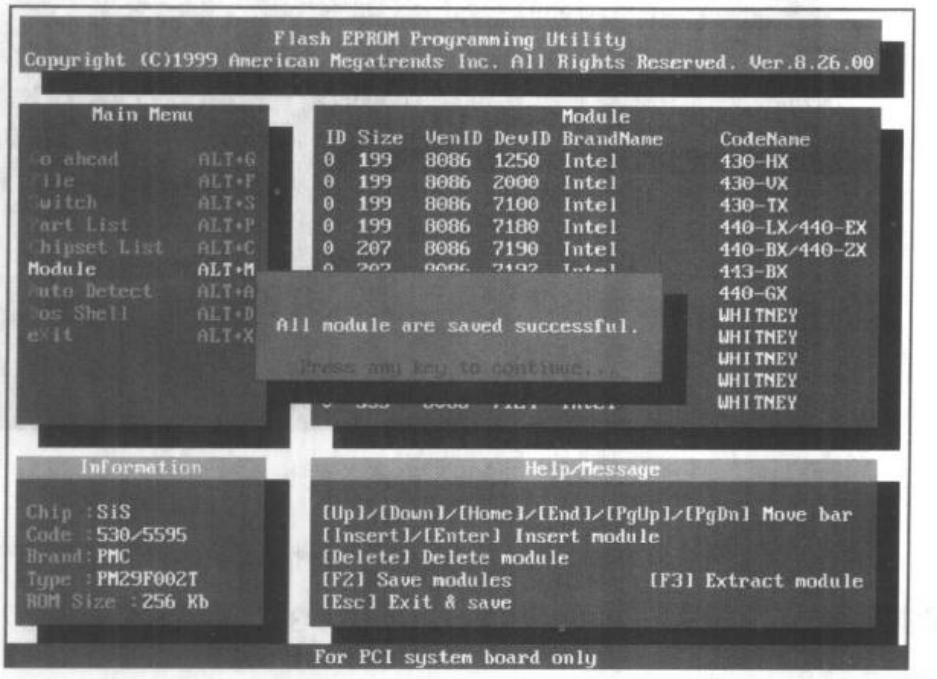


图 8-30 将删除掉 430FX 模块数据保存 AMIFLASH.EXE

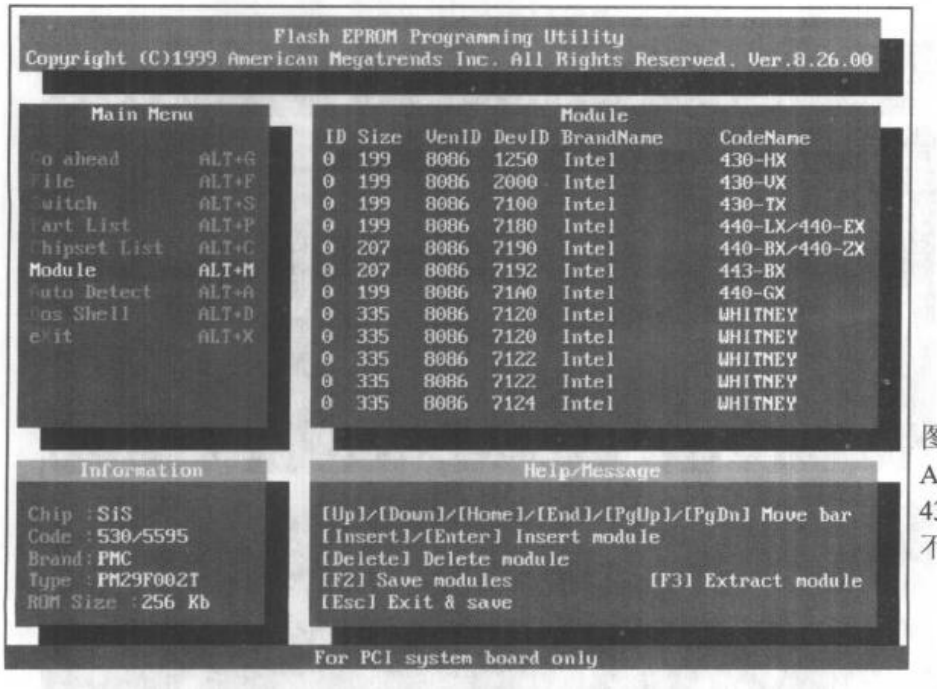


图 8-31 重新加载 AMIFLASH.EXE，430FX 模块数据已不存在了

我们可以检查一下，目前的目录下确实多出430FX.FMF 这个文件(如图8-32 所示)，文件长度为 207 bytes。

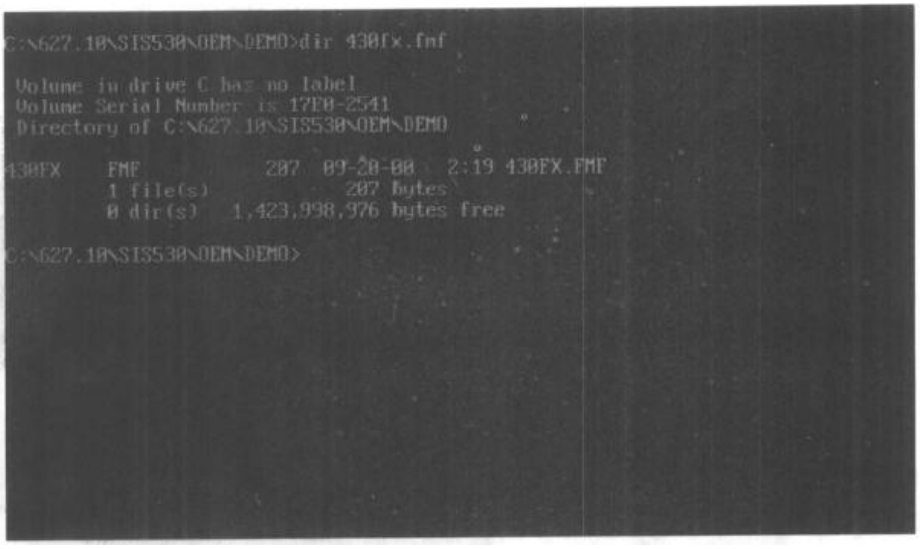


图 8-32　430FX.FMF 文件长度为 207 bytes

现在我们要进行插入新模块(此处以刚刚分离的430FX.FMF充当)的操作，在欲插入的字段前按下[redacted]或[redacted]键(如图8-33)，然后输入要插入的模块文件名称(如图 8-34 所示)。

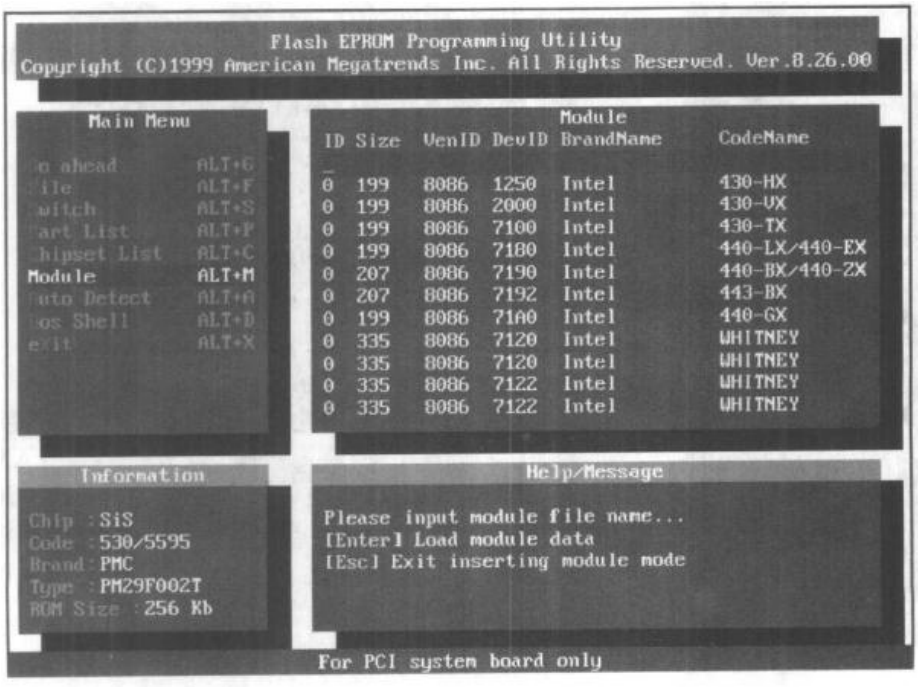


图 8-33　在欲插入的字段前按下[redacted]或[redacted]键

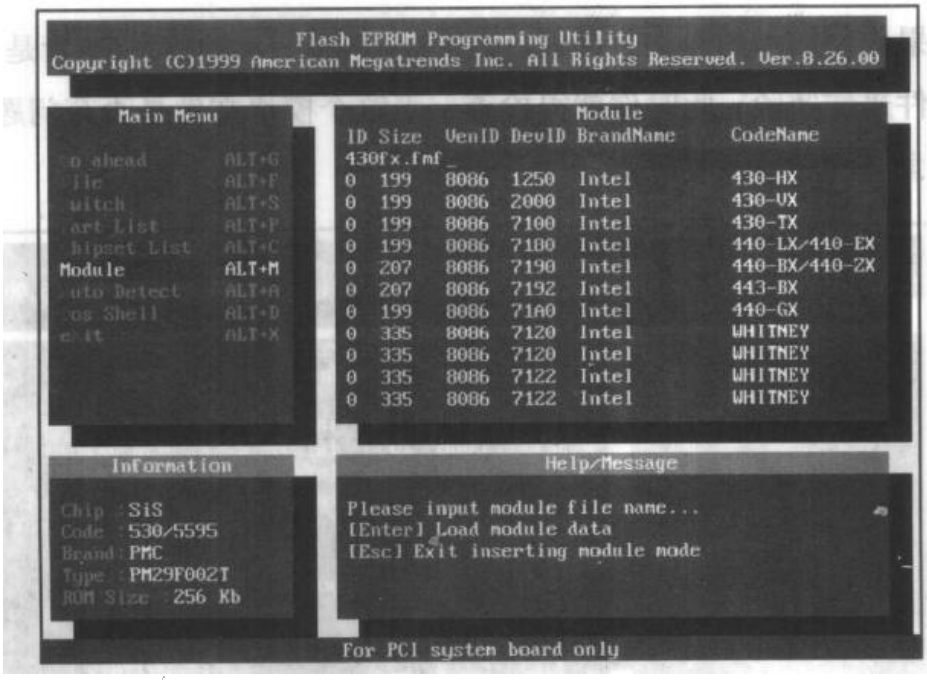


图 8-34　输入所要插入的模块文件名称

输入模块文件名后确定无误，430FX(或新的芯片组／快速存储器芯片型号)会重新纳入模块数据库(如图 8-35)，此时只要记得按 ■ 保存就行了。

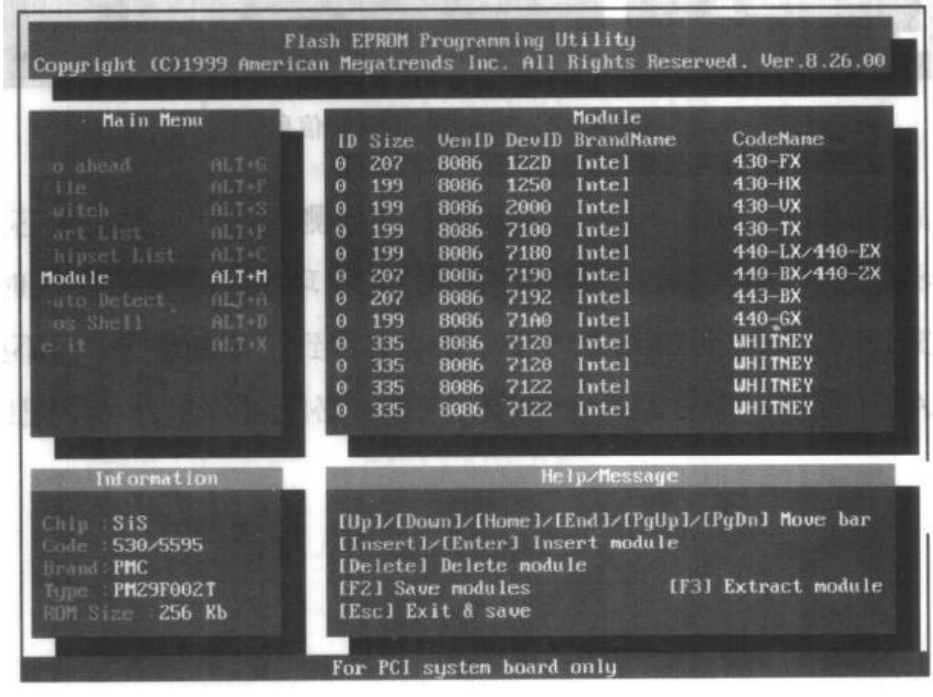


图 8-35　430FX 重新纳入模块数据库

如果你所插入的模块有问题，会出现图8-36所示的信息(笔者是故意损坏这个文件来作实验)，此时你就得检查一下这个模块文件是否有问题，或者向AMI 公司寻求技术支持了。

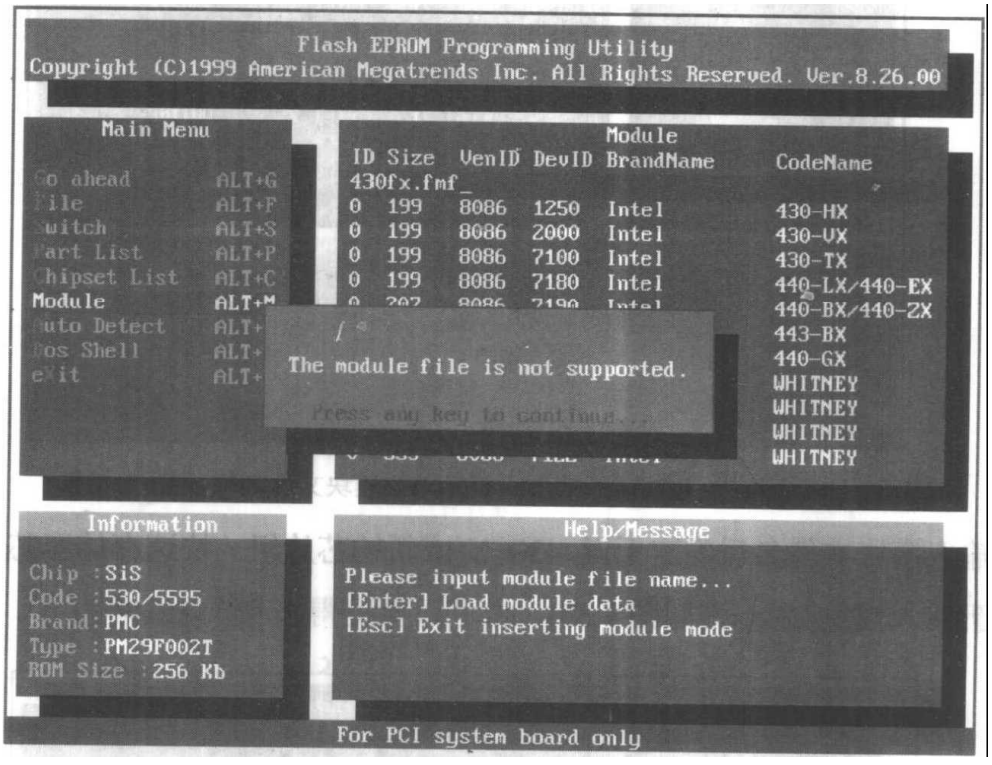


图 8-36　有问题时的提示信息

最后有三个主选项，Auto Detect 是自动侦测当前计算机主板芯片组与快速存储器型号；Dos Shell 则是暂时跳到 DOS 环境，以便于你临时找不到、抓不到文件时，先切到DOS来搜寻、移动文件到正确的目录，而不必先退出AMIFLASH 程序；最后的 Exit 选项，应该不必笔者再多介绍了吧！

8-3 BIOS 组态魔术师——AMIBCP.EXE 之一

如果说 AMIFLASH.EXE 是笔者所看过的最强大的 FlashEPROM 烧写程序，那么 AMIBCP.EXE 这个 BIOS 模块组态设置的工具程序，简直可以用出神入化加以形容。我们可以先输入 AMIBCP(或 BCP7B12)执行这个程序，语法是：

```
amibcp [biosfile.rom]
```

会出现如图 8-37 所示的画面。

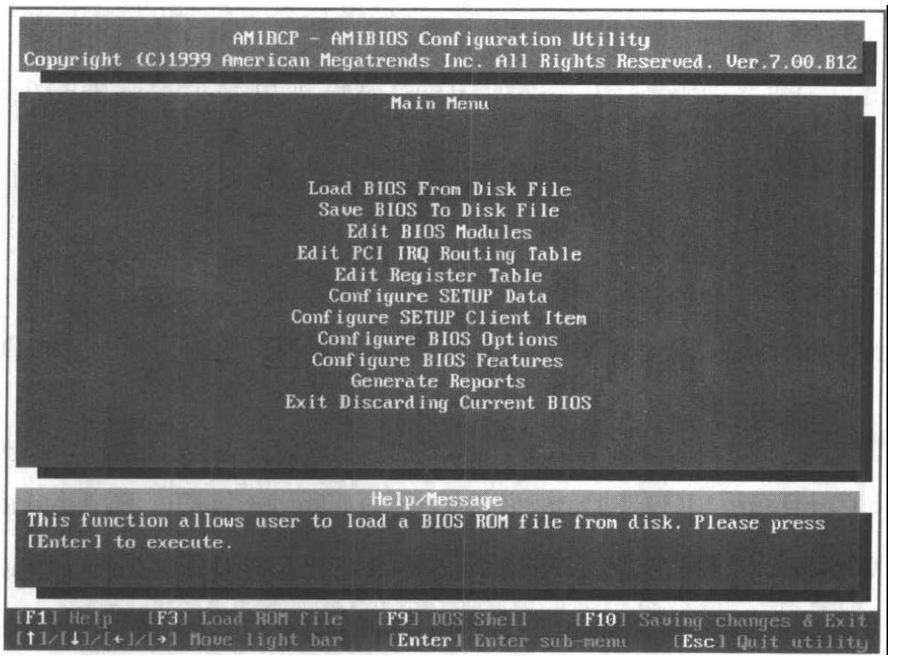


图 8-37 AMIBCP.EXE 的执行程序

我们看到的是 AMIBCP 第一层主选项，它有下列 11 个项目：

① Load BIOS From Disk File (加载BIOS 影像文件)

② Save BIOS to Disk File (将修改过的BIOS 组态保存到BIOS 影像文件)

③ Edit BIOS Modules (编辑 / 修改BIOS 影像文件内的模块)

④ Edit PCI IRQ Routing Table (编辑PCI IRQ 绕线参数设置表格)

⑤ Edit Register Table (编辑寄存器填值表格)

⑥ Configure SETUP Data (SETUP的画面选项开启 / 关闭与默认值组态)

⑦ Configure SETUP Client Data (SETUP Client 设置客户端的数据组态设置)

⑧ Configure BIOS Options (BIOS 错误提示、硬件项目显示等选项的组态设置)

⑨ Configure BIOS Features (BIOS信息检视与预设密码 / 厂商字符串组态设置)

⑩ Generate Report (产生 BIOS 详细信息的报告文件)

⑪ Exit Discarding Current BIOS (放弃目前的 BIOS 组态设置，回到DOS)

用[redacted]、[redacted]键移动控制光条到想选择的主项目，按下[redacted]就可以进入；而像第六项 Configure SETUP Data，底下还会有 Standard SETUP、Advanced CMOS Setup、 Chipset SETUP、 Power Management SETUP、 PCI/PnP Setup、 Peripheral SETUP、Hardware Monitor SETUP，一共七个第二层次项目，所以有关第六项 Configure SETUPData 部分，我们将在稍后的章节，以及第 7 章 BIOS 画面、选项与默认值设计作补充介绍。

如果我们在 AMIBCP 后面没有指定 BIOS 文件名，AMIBCP 会出现一个对话窗口，要求你输入 BIOS 影像文件名，此时你输入正确路径的 BIOS 影像文件名，如图 8-38 所示。

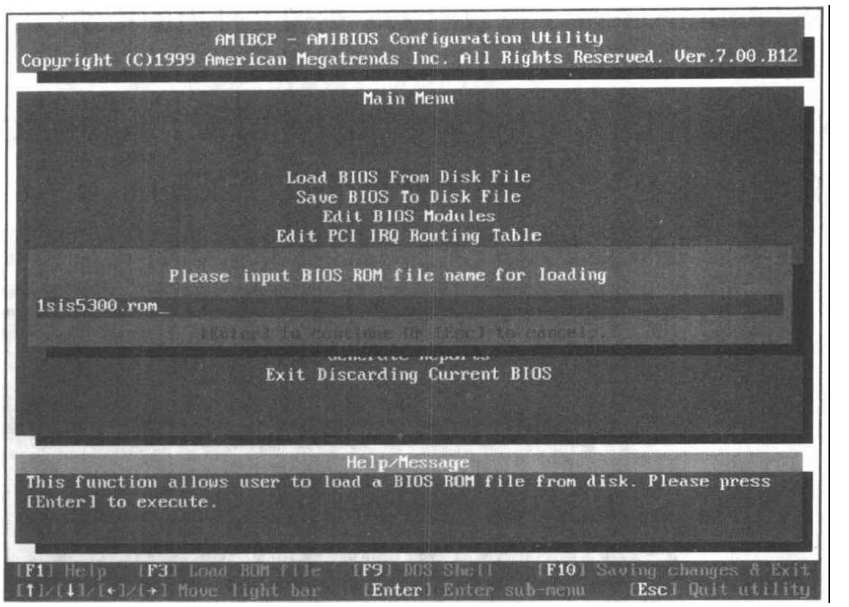


图 8-38　输入正确的 BIOS 影像文件名

如果修改好的 BIOS 文件要保存，同样可以移动光条到 Save BIOS to Disk File 的字段(或者按 [redacted])，然后以先前的文件名称存储起来；如果要另外存个文件名，就以光标向左清除旧的文件字符串，输入新的名称即可。

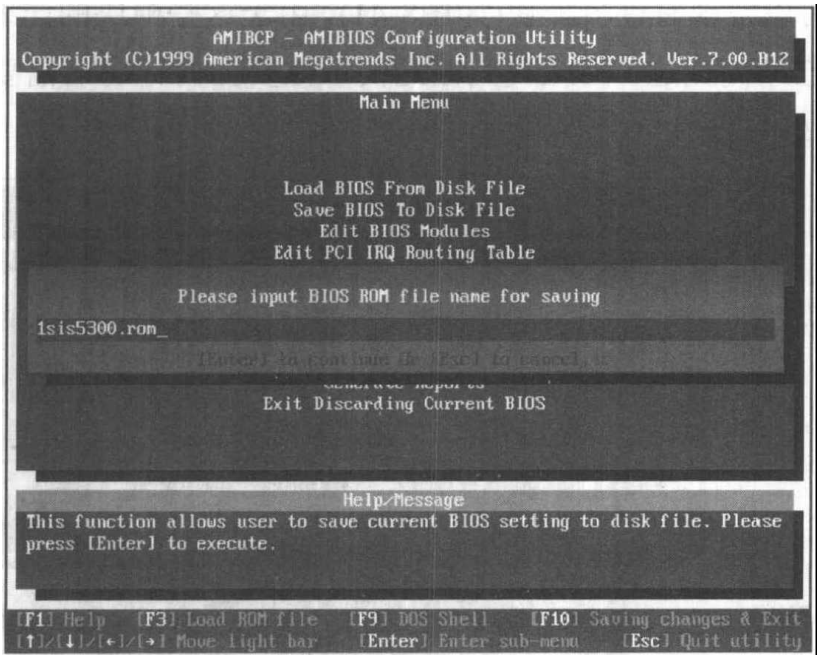


图 8-39　回车保存

Edit BIOS Modules(编辑 / 修改 BIOS 影像文件内的模块)

以本书光盘所附的文件为例，我们用 AMIBCP 1SIS5300.ROM 打开这个porting好的 BIOS 影像文件，然后选择 Edit BIOS Modules (如图 8-40 所示) 。

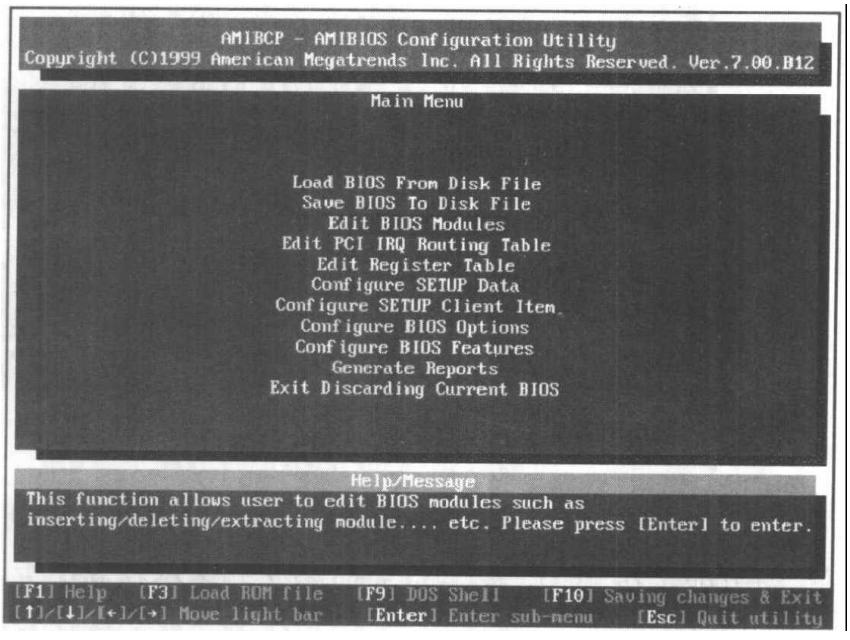


图 8-40　选择 Edit BIOS Modules

我们可以发现有许多模块(如图8-41)，由于这些模块的详细定义在第2、3 章都已经提到过，因此我们在此仅示范要如何针对所需要的模块作修改、删除的动作。

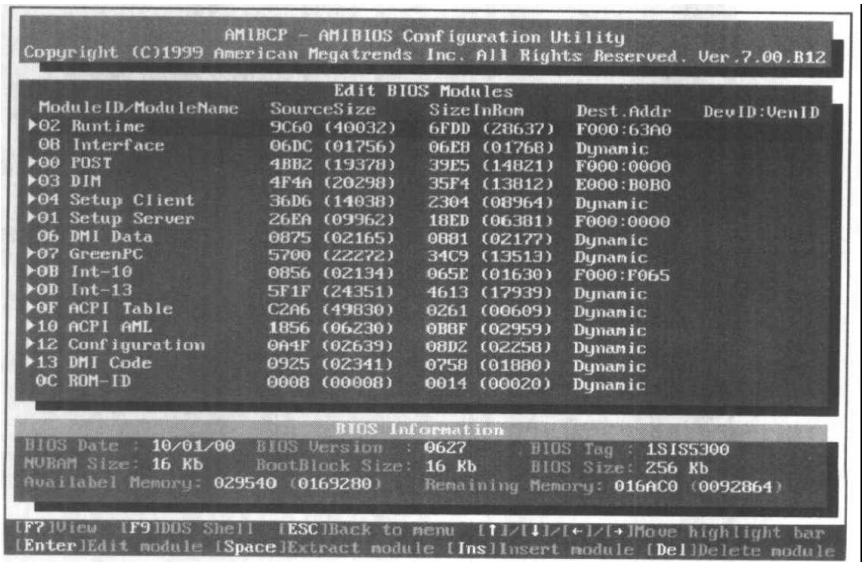


图8-41　Edit BIOS Modules 下的模块

AMIBIOS 内部模块 ID、可更动特性与模块用途

模块名称	ID	可否修改	模块用途
POST	00h	否	POST 程序代码
Setup Server	01h	否	AMIBIOS 设置服务端
Runtime	02h	否	BIOS 功能函数 / 常驻的程序代码
DIM	03h	否	Device Initialize Manager 装置初始化
Setup Client	04h	可	提供 AMIBIOS 设置画面的界面程序
Remote Server	05h	可	搭配远程遥控除错系统 (Remote Debugger)
DMI Data	06h	可	Desktop Management Inteface 数据区
GreenPC	07h	否	符合 APM 规范的节电管理程序
MPS	09h	可	多重处理器 (Multi-Processor, MP) 起始模块
Int-10	0Bh	可	BIOS 文字 / 绘图等显示处理模块
Int-13	0Dh	可	BIOS 磁盘读写模块
ACPI Table	0Fh	可	ACPI规范的控制表格区
ACPI AML	10h	可	ACPI 之 AML 控制码存放区域
P6 Micro Code	11h	否	Intel P6 处理器微码修补模块
Configuration	12h	可	系统组态显示模块
DMI Code	13h	可	DMI 控制程序码模块
PCI AddOn ROM	20h	可	PCI 适配卡 ROM 程序代码模块
Multilanguage	21h	可	适配卡 ROM 程序代码模块
Font DataBase	30h	可	中文 (双位语系) 字形资料模块
OEM Logo Data	31h	可	屏幕右上方 OEM 商标画面模块
Graphic Logo Code	32h	可	Silent Boot, OEM 全屏幕画面处理程序
Graphic Logo Data	33h	可	Silent Boot, OEM 全屏幕图形数据模块
Virus	36h	可	趋势 Trend Chip Away Virus (TCAV) 模块

基本上的按键方式是通过↑、↓键来移动控制光条，在可卷绕的列举模块中选择要更动的模块，按Enter就可直接进入修改(可供修改的模块，请参考上表所示)；如果要删除某一个模块则按Delete键；想要解压这个模块，并且另外存一个文件，则按下Space之后，输入一个代表这个模块的文件名称就行了；若是要安插一个外加模块，则按下Insert键之后，输入该模块的完整路径与文件名称就可以了。

举个例，我们想把所附光盘内，计划目录(\627.10\SIS530\OEM\DEMO)下的1SIS5300.ROM，这是一个已经编译妥当的BIOS ROM影像文件，它的原始默认值是以HF122.BIN作为Setup Client(设置界面服务程序)，我们示范一种不需要重新修改OEM.MAK 、Makebios 或 Makeall 重头编译、链接，直接以AMIBCP.EXE换掉这个界面模块的方法。

首先输入AMIBCP 1SIS5300.ROM，以AMIBCP打开这个1SIS5300.ROM文件。接下来我们选择 Edit BIOS Module，然后卷到最底下一行 (如图8-42所示)，按下Insert键之后，输入要添加的模块，此处我们指定 SP120.BIN，也就是 Simple Setup 的界面程序模块，如图8-43所示。

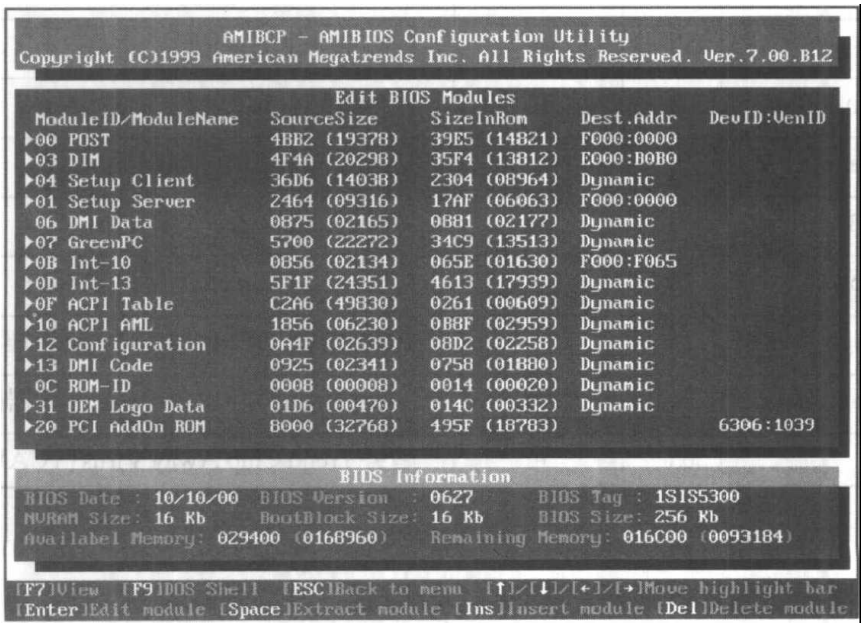


图8-42　选择Edit BIOS Moudle的最底下一行

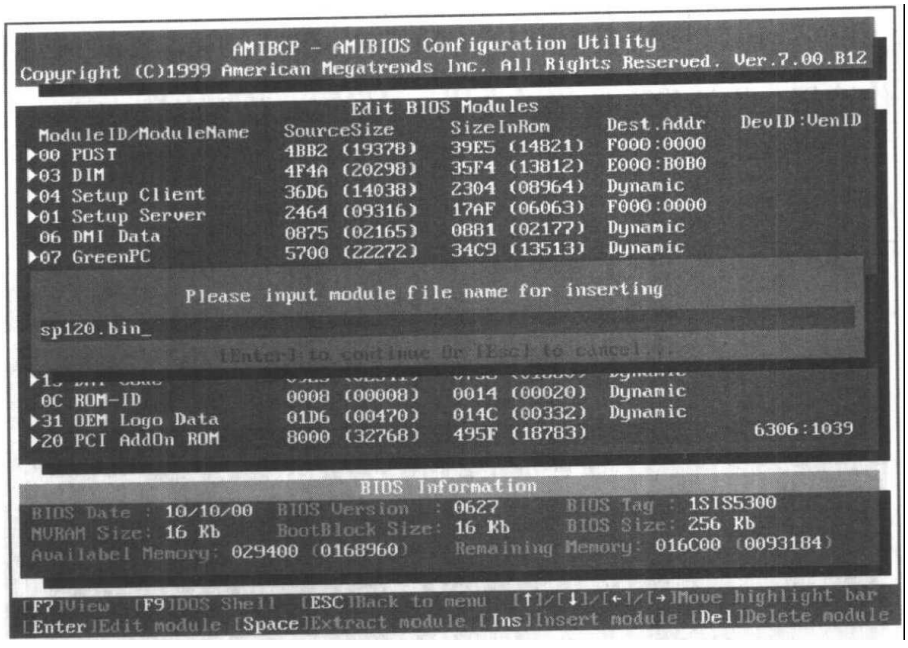


图 8-43　输入要添加的模块名

插入 SP120.BIN 这个文件成为 BIOS 模块之一，此时 AMIBCP 会出现图 8-44 所示的模块微调画面。

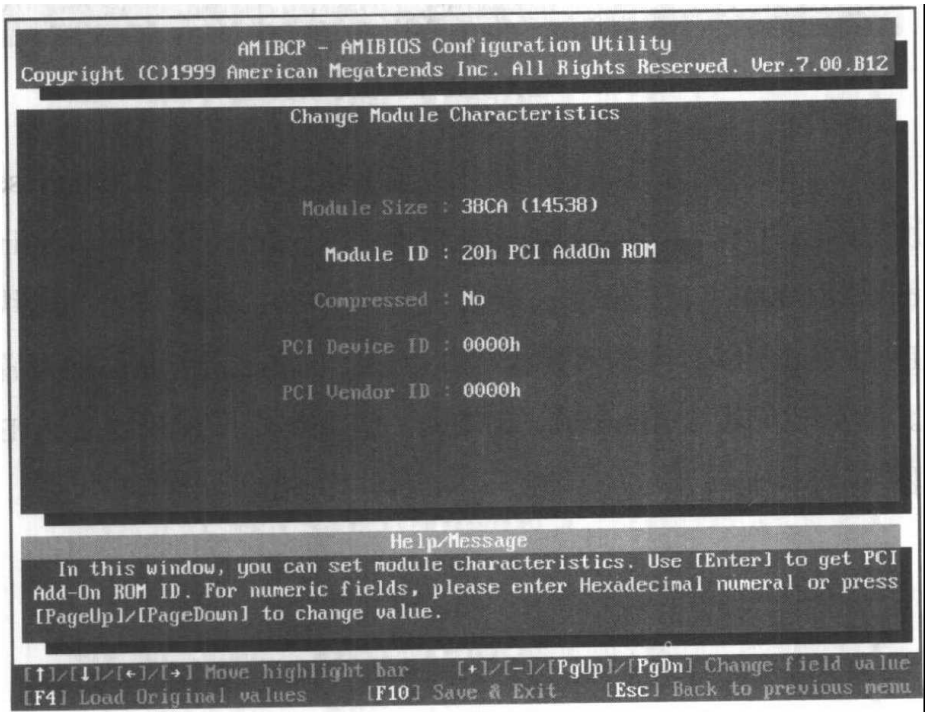


图 8-44　模块微调画面

此时我们就可以按照我们所插入的模块属性，用Page up、Page Down键或←、→键来作选择。此处我们选择04 Setup Client(设置界面程序模块)，如图8-45所示。

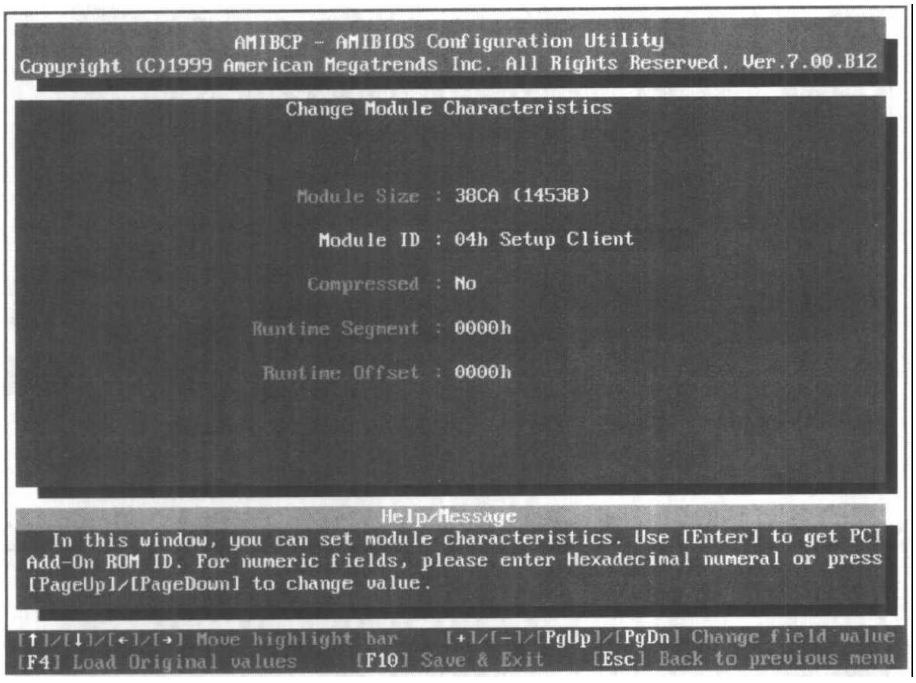


图8-45 选择设置界面程序模块

接下来我们还可以设置的项目有是否先压缩再加入模块(Compressed设Yes)，此模块预定解压执行的内存段地址与偏移地址(Runtime Segment，Runtime Offset)，这两者都设为0000代表采取动态配置执行的方式；若所加入的模块是某PCI DeviceID / VendorID专用的监控程序数据，也可以在这个画面做设置(限ID为20h时)。此处我们设Compressed为Yes，也就是先压缩后再加入BIOS模块内，如图8-46所示。

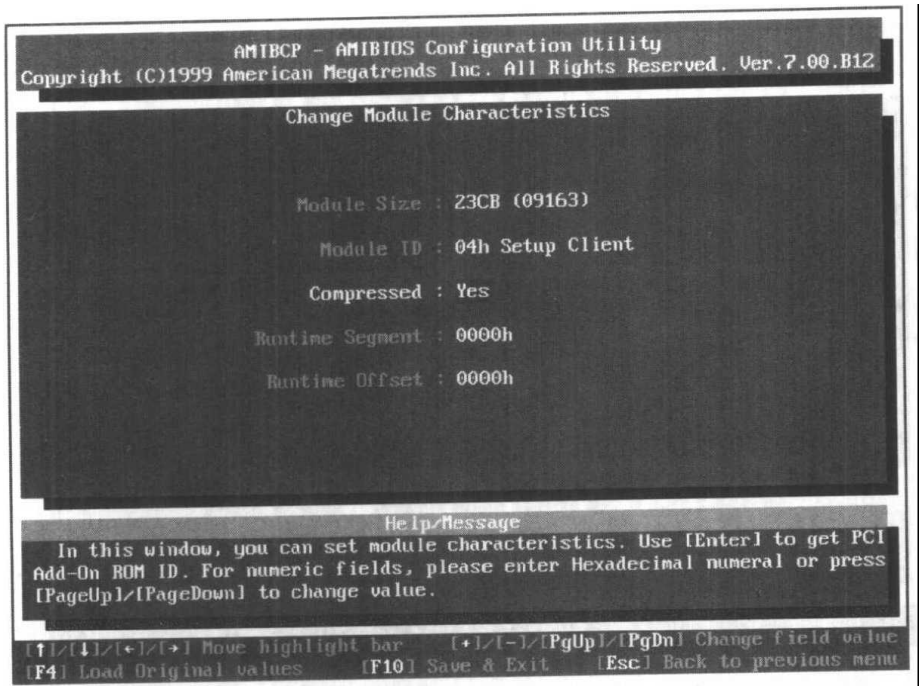


图 8-46 设置 Compressed 为 Yes

接下来记得按 [redacted] 存储组态，它会出现如下图的信息，按下 [redacted] 后组态写回并回到上一个画面，此时我们可以发现，真的多了一个新的 Setup Client (ID=04)模块(如图 8-47)，源文件长度为 14538bytes(正是 SP120.bin 的源文件长度)，压缩过后的长度为 9175bytes(如图 8-48 所示)。

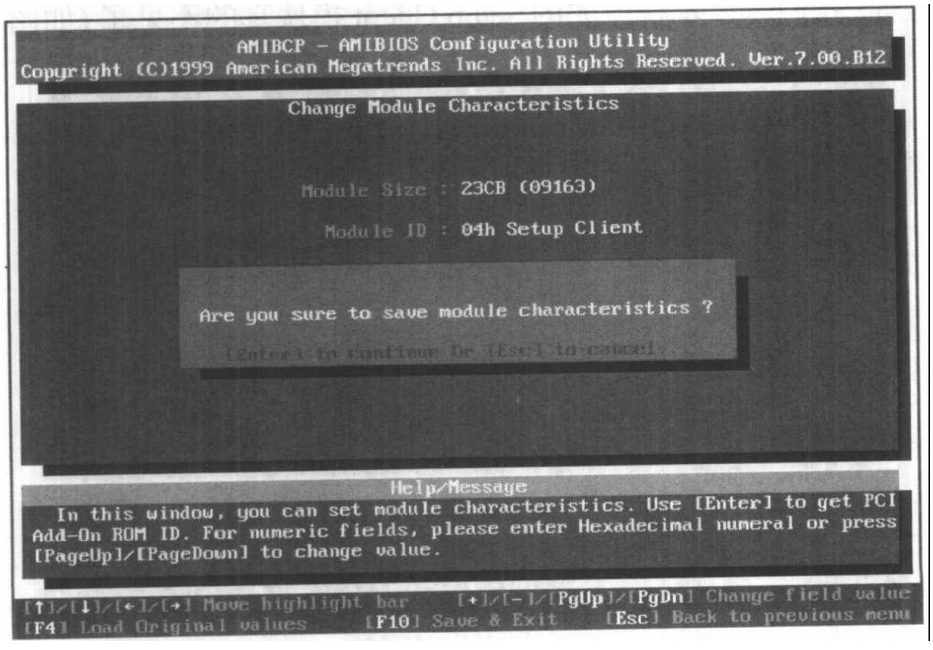


图 8-47 多了一个新的 Setup Client 模块

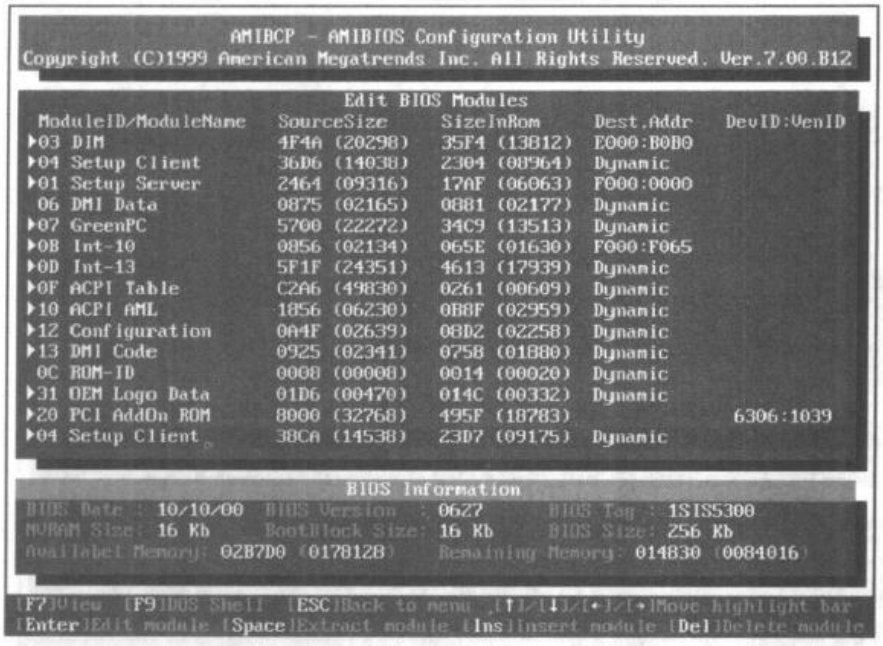


图 8-48　源文件长度为 14538bytes，压缩过后的长度为 9175bytes

由于 AMI BIOS 特性就是从头找第一个 ID=04的模块作为 Setup Client，即使 BIOS 存在两份 ID 为04的 Setup Client 模块，因此我们还必须把上面第一个ID=04的旧Setup Client程序模块给删除掉，此时我们可以将光标选到旧的Setup Client 模块那个字段，按下[redacted]键来删除（如图 8-49），确定删除的动作无误后按[redacted]键后，我们可以发现原先的Setup Client 模块真的不见了 (如图8-50)。

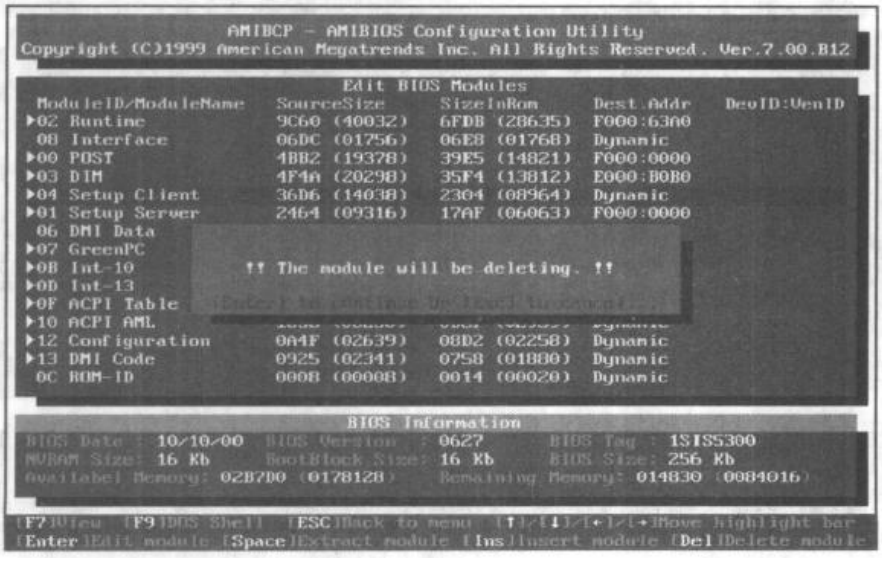


图 8-49　将第一个旧的 ID=04 的 Setup Client 模块予以删除

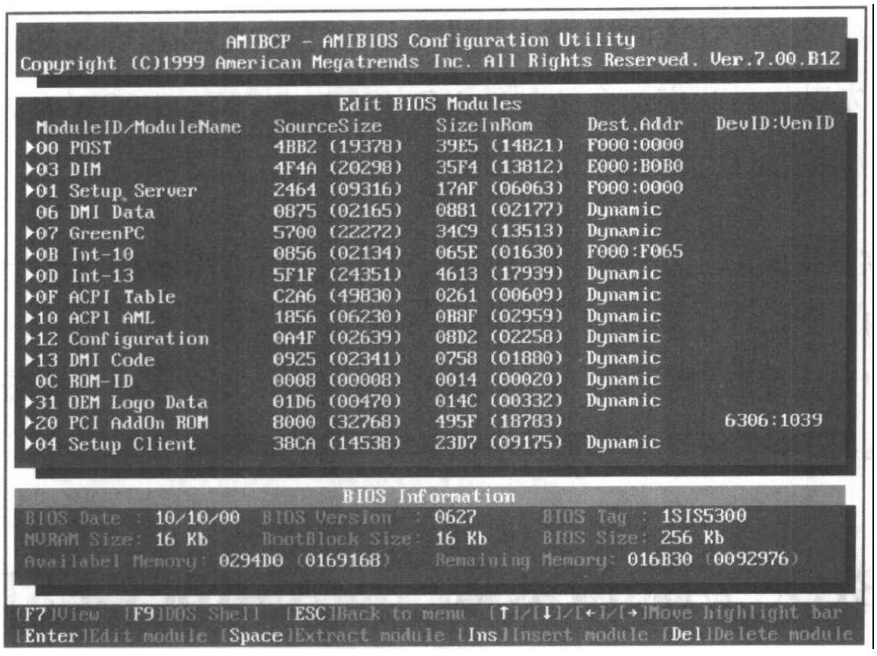


图 8-50 保留新的 ID=04 SetupClient 模块

模块删减、修改过后，仍需要回到第一层主菜单画面，选择 Save BIOS to DISK File 的选项字段，此时按下 [redacted] 并输入一个新的文件名(如图8-51)，组态过的新 BIOS 文件就这样生成了。

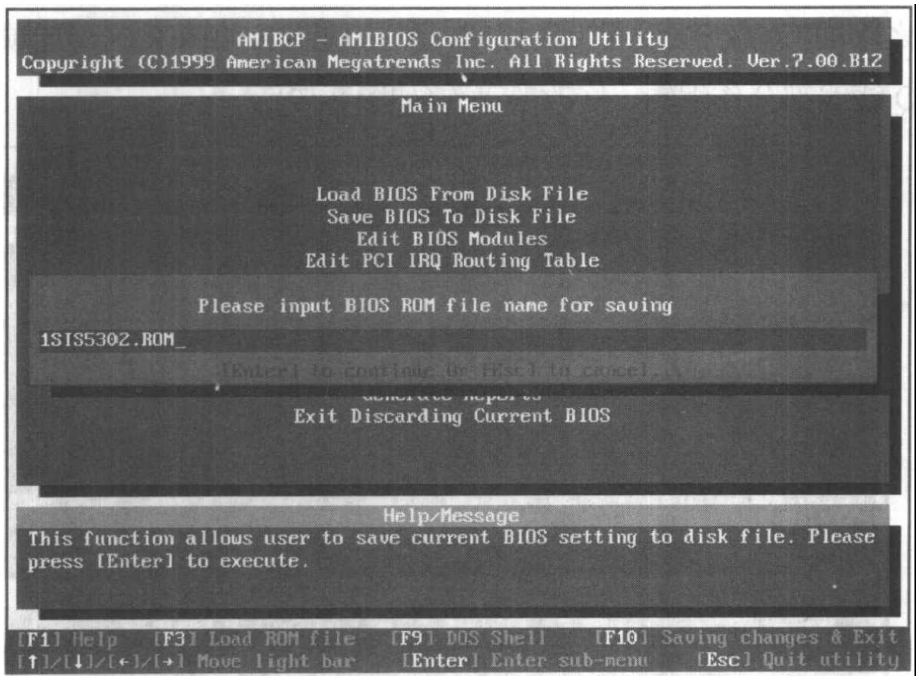


图 8-51 予以存储并给予新的文件名

此时你可以把1SIS5302.ROM直接以AMIFLASH.EXE写入FlashEPROM内存，或者先以稍后我们会介绍到的ROMSETUP.EXE，SHOWBIOS.BAT先测试一下，确定无误再写到FlashEPROM。

如果读者汇编语言功力了得，其实也可以一次就把四种接口程序模块都加入BIOS内，然后通过第7章SSP语言的编写，以及内部的部分程序代码来改写，做出连设置的界面程序也可以选择的特殊BIOS，这部分就留给读者自行发挥与运用了！

8-4 BIOS组态魔术师——AMIBCP.EXE之二

Edit PCI IRQ Routing Table (编辑PCI IRQ绕线参数设定表格)

如果我们Porting出来的BIOS，开机时Check point(debug code)停留在2Ah、38h，这多半是表示主板总线启动不正常，或者外围组件(PCI音效芯片、SuperI/O芯片)之间的IRQ绕线设置没调好时，就是通过AMIBCP这个选项，来针对硬件IRQ绕线部分作编辑、修改的动作(如图8-52)。

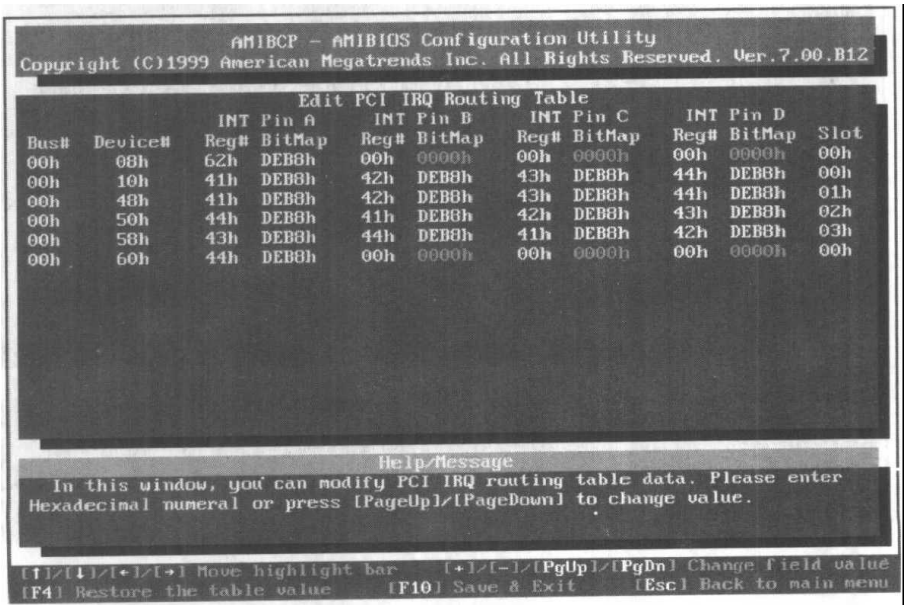


图8-52 通过AMIBCP来针对硬件IRQ绕线部分作编辑、修改

由于读者在 Porting BIOS 时，或多或少会有很大的机会需要用到 PCI IRQ Routing，因此这个部分我们会做仔细一点的说明。首先我们选 Edit PCI IRQ Routing Table，它会出现如图 8-53 所示的多重选项 / 字段画面。

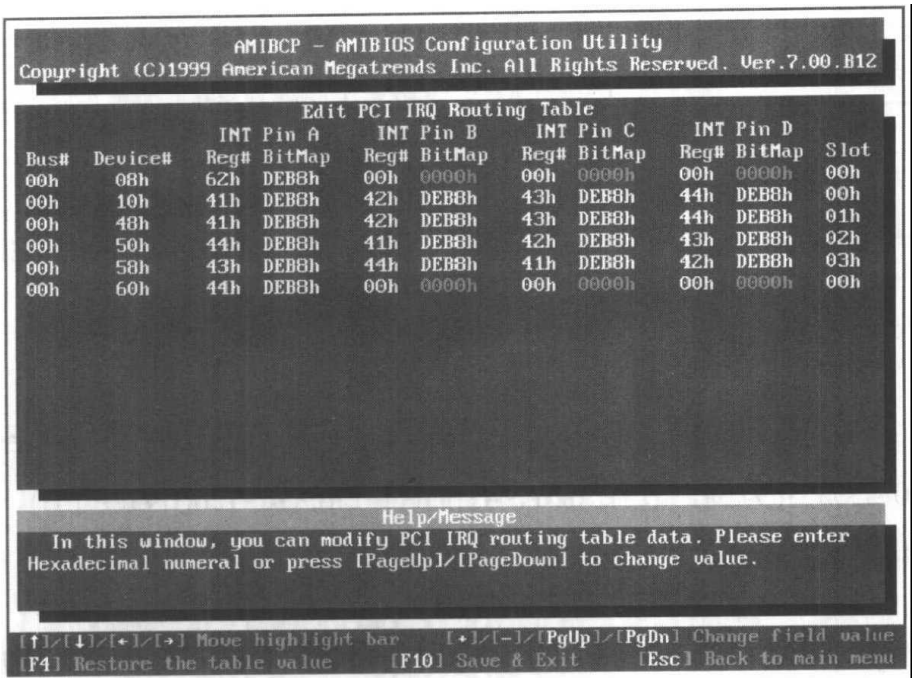


图 8-53　多重选项 / 字段画面

Bus#：所指的是 PCI Bus Number，从 0～1Fh。由于绝大多数主板只实作单一 PCI 总线，这个字段应该设为 0；若主板有多重 PCI Bus 的设计，或者有 PCI 桥接器、桥接芯片的设计，则视该装置 / 扩充槽隶属于哪一组 PCI Bus 总线号码，填入对应的数值。

Device#：该 PCI 扩充槽或装置号码 *8 的数值。有关扩充槽与装置号码转换规则如下表：

PCI扩充槽ID选择与装置号码查询表

ADsel	AD11	AD12	AD13	AD14	AD15	AD16	AD17	AD18	AD19	AD20	AD21
Device	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0 A h
Device#	00h	08h	10h	18h	20h	28h	30h	38h	40h	48h	48h
ADsel	AD22	AD23	AD24	AD25	AD26	AD27	AD28	AD29	AD30	AD31	
Device	0Bh	0Ch	0Dh	0Eh	0Fh	10h	11h	12h	13h	14h	
Device#	58h	60h	68h	70h	78h	80h	88h	90h	98h	A0h	

Onboard 装置以该装置硬件线路对外拉出的 ADsel 号码，减去 11 之后的值为 device 号码，Device 号码限制在 00～1Fh 之间，连同 PCI 扩充槽最多只能有 32 组 PCI 装置。

Reg#：IRQ 中断对应寄存器号码。每一家芯片组的寄存器号码各有不同，以 SIS530 芯片组为例，使用 62h、41h(INT PinA)、42h(INT PinB)、43h(INT PinC)与 44h(INT PinD)。

BitMap：指定可以分配使用的 IRQ 号码，此字段由 16 个字节组成，每个字段位设为 1，代表该字段对应的 IRQ 号码可以加入分配使用；=0 则禁止使用该 IRQ 号码。

例如：

DEB8h=1101 1100 1011 1000b，第 15、14、12、11、10、7、5、4、3 个位字段都设为 1，因此指定这个装置可以选用 IRQ15、IRQ14、IRQ12、IRQ11、IRQ10、IRQ7、IRQ5、IRQ4 与 IRQ3 这些中断，并通过 Plug And Play(PnP)的方式，跟其他的装置一起 Share 或者独占。

IRQ_Maskbit

1	1	0	1	1	1	0	0	1	0	1	1	1	0	0	0
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Slot：该绕线设定字段所对应的 PCI 扩充槽号码数，从 1～6。

基本上设定方式一样，也是通过←、→键移动字段，Page up、Page Down键或↑、↓键来调整数值。例如图 8-54、图 8-55 所示范的把第一列中 Device# 字段设成 10h 的做法。

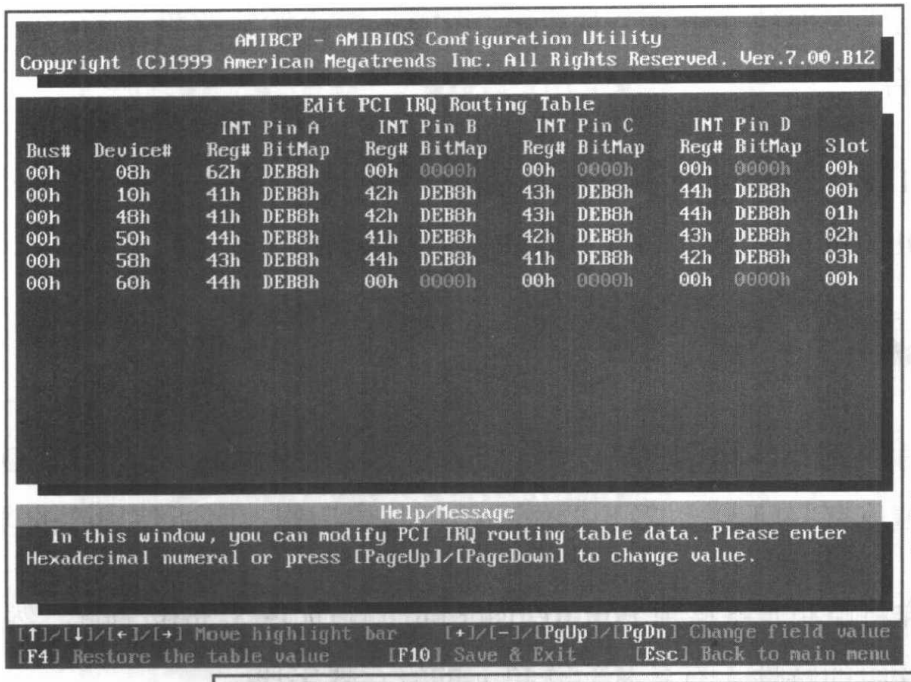


图 8-54 第一列 Device# 字段为 08h

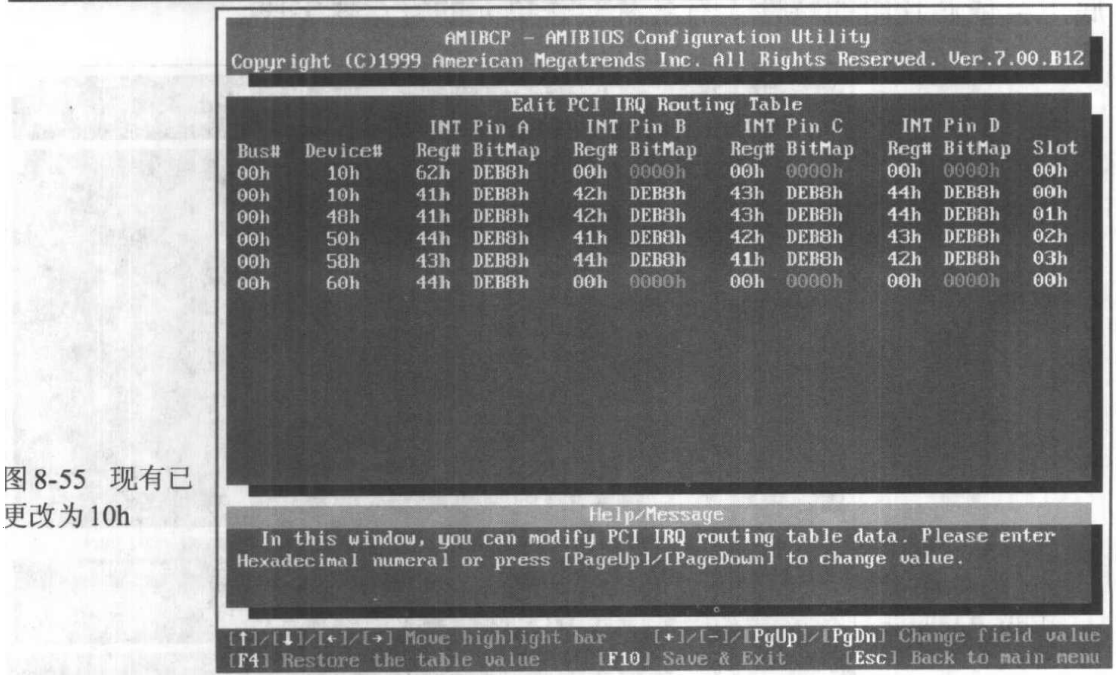


图 8-55 现有已更改为 10h

由于PCI的IRQ绕线设置，必须借助该型主板原始设计的布线蓝图(PCB Layout)，查寻做在主板上的每个PCI装置、扩充槽等，是接哪一组ADsel nn？若是PCI扩充槽，还要多考虑该扩充槽的INTA、INTB、INTC、INTD与真正的PIRQ0、PIRQ1、PIRQ2、PIRQ3的线路连接对应关系，所以一个BIOS要能成功地被移植到某块主板，需跟主板原始线路设计的研发人员有密切的配合。

有关PCI IRQ Routing的其他相关章节，可参阅前面几章讨论到RUNPCI.ASM以及OEMPORT目录下的OEMRPCI.ASM的部分。

Edit Register Table

修改芯片组寄存器起始表格的用途，是你发现所Porting的这组BIOS对硬件寄存器填值有些问题，以及想要做些检视与事后修改时所使用到的项目。

我们一样将控制光条移到Edit Register table段(如图8-56)按[redacted]，然后它底下会依芯片组的特性，有各种运作单元的寄存器字段。

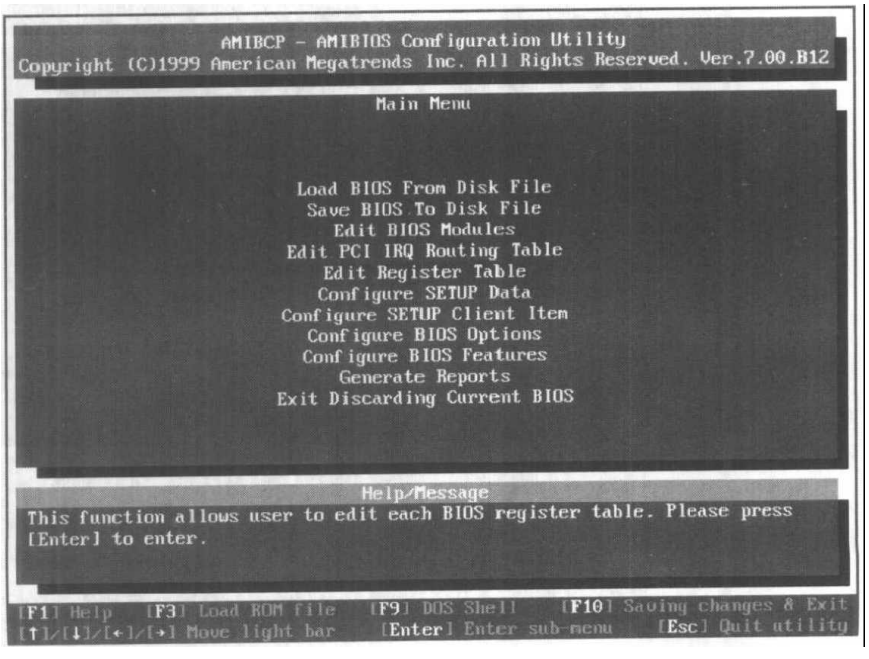


图8-56 将控制光条移到Edit Register table

我们以Sis530芯片组为例，它共有六项运作单元 / 寄存器字段(如图8-57)，我们想要检查其中的SIS5513 IDE Reg init，也就是关于IDE部分寄存器的设置，则用 [↑]、[↓] 选到 SIS5513 IDE Reg init 字段，按 [Enter] 后出现如图8-58的画面，此时我们就可以做检查、修改的动作，事后有必要另外存一个新的.ROM文件就成了。

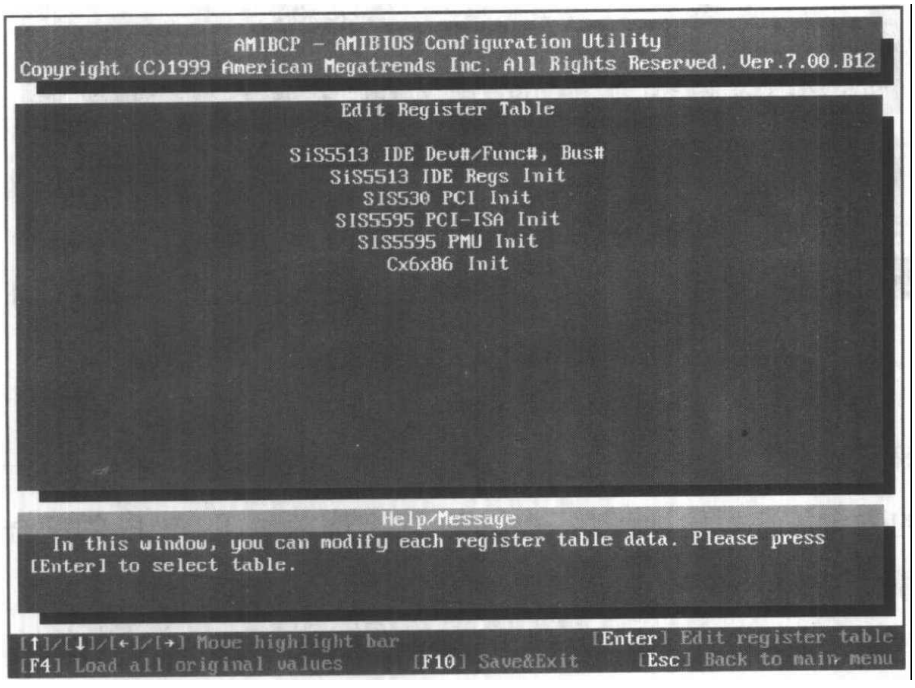


图 8-57　Sis530 芯片组的六项运作单元 / 寄存器字段

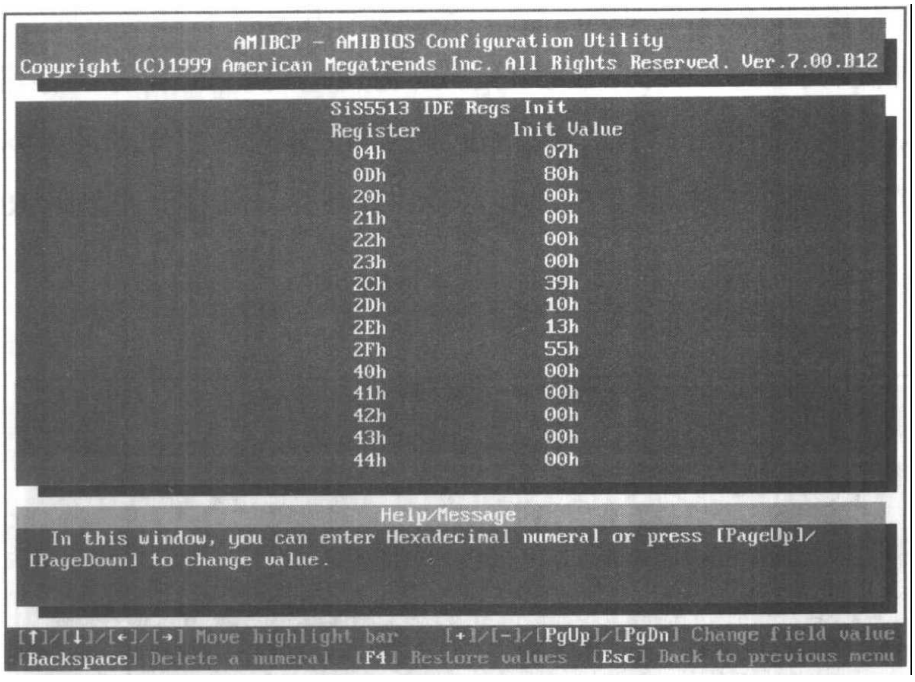


图8-58　选中SIS5513 IDE Reg init字段后出现的画面

对于硬件寄存器的检视与修改，在稍后的第9章中我们还会介绍一套称为RU.EXE 的强大寄存器检视程序，AMIBCP 这个选项与 RU.EXE 的配合，可以说是如虎添翼。

CHAPTER

9

AMI BIOS 工具程序
——研发 / 维护工具

所谓“工欲善其事，必先利其器”。为了让整个 BIOS 源代码在维护以及移植作业上的顺利，AMI 公司开发了许多有助于 BIOS 模块修改、删减、烧写，以及其他特定用途的工具程序。

前面介绍了偏ROM烧写与模块编辑的工具程序，我们除了将功能未介绍完毕的 AMIBCP.EXE 再做最后的介绍之外，也针对其他 BIOS 辅助性研发工具、维护工具程序作一下介绍。

9-1 BIOS 组态魔术师—— AMIBCP.EXE 之三

Configure SETUP Data(设置数据项的组态)

Configure SETUP Data是唯一位于主选项底下，有七个次选项画面的两层次选项结构(如图9-1)，这个字段也是针对BIOS ROM影像文件的每个设置画面、提问选项、默认值与开机数值等重要的主选项。

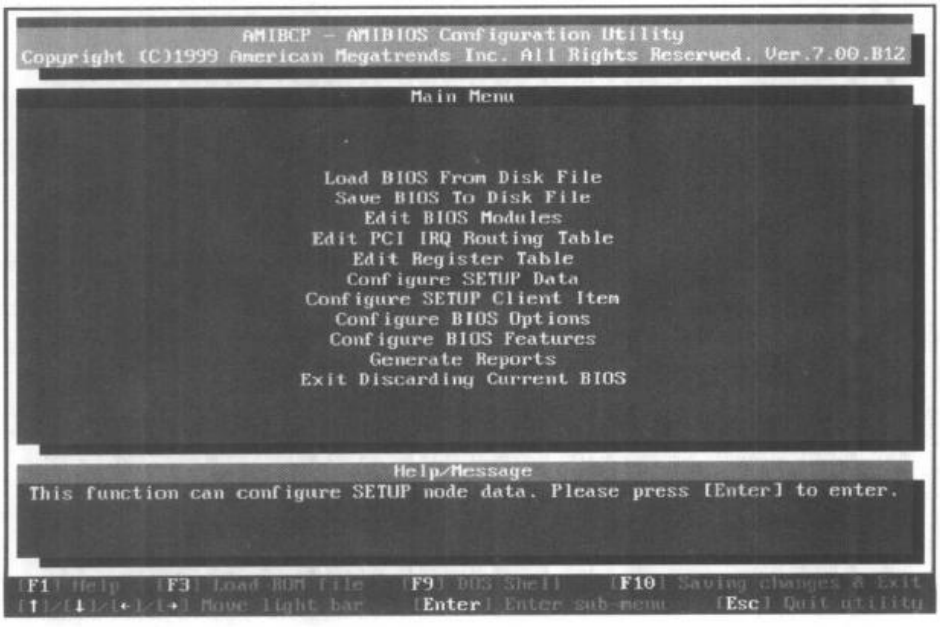


图9-1 AMIBCP的主菜单

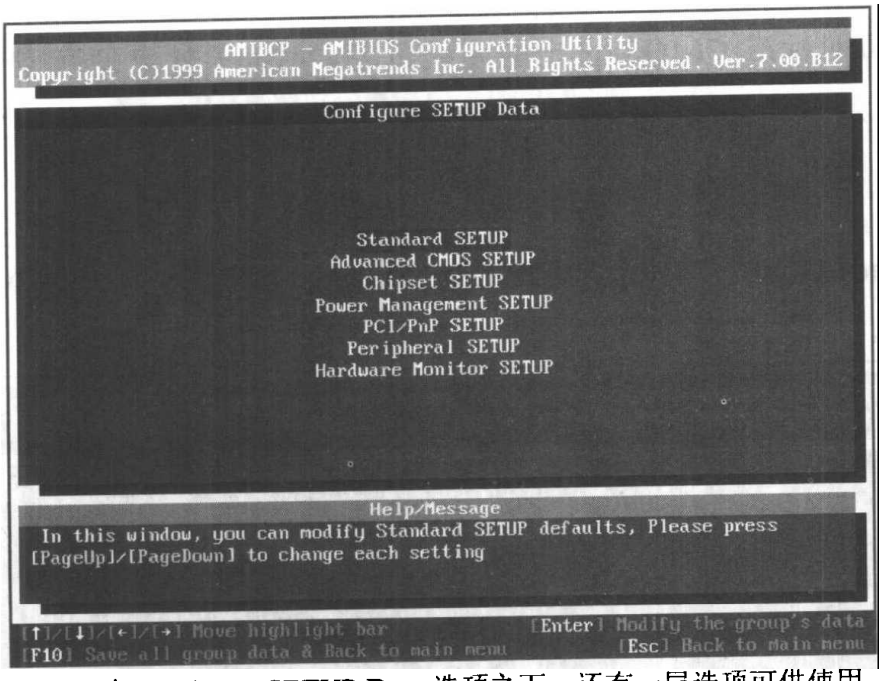


图 9-2　在 Configure SETUP Data 选项之下，还有一层选项可供使用

每个字段的组设定方式，跟前面介绍过的画面相同，同样用[→]、[←]左右方向键来移动字段，[PageUp]、[PageDown]键或[+]、[-]键来调整数值。

首先介绍这个选项字段下有哪些第二层选项项目。

① **Standard SETUP:** 标准CMOS设置。进入后会出现如图9-3 所示的设置画面。

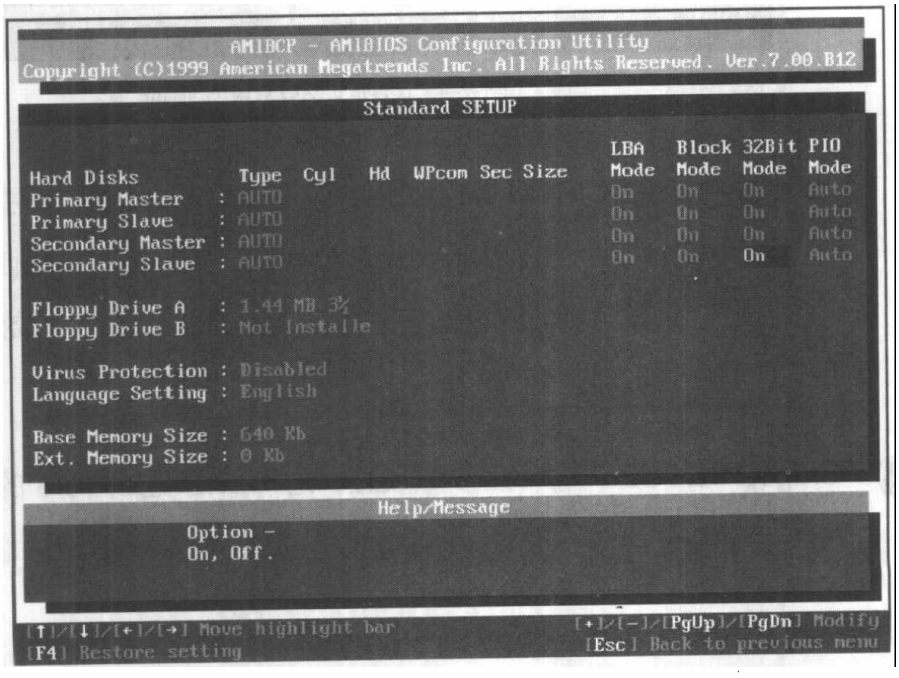


图9-3　标准CMOS 设置画面

此时可对标准 CMOS 设定的系统日期、时间、软盘硬盘型号进行设定，以及开机防毒功能的开启 / 关闭功能预先调整状态，然后保存到 BIOS 文件中。

② **Advanced CMOS Setup：** 进阶 CMOS 设置。进入后出现如图 9-4 的设置画面。

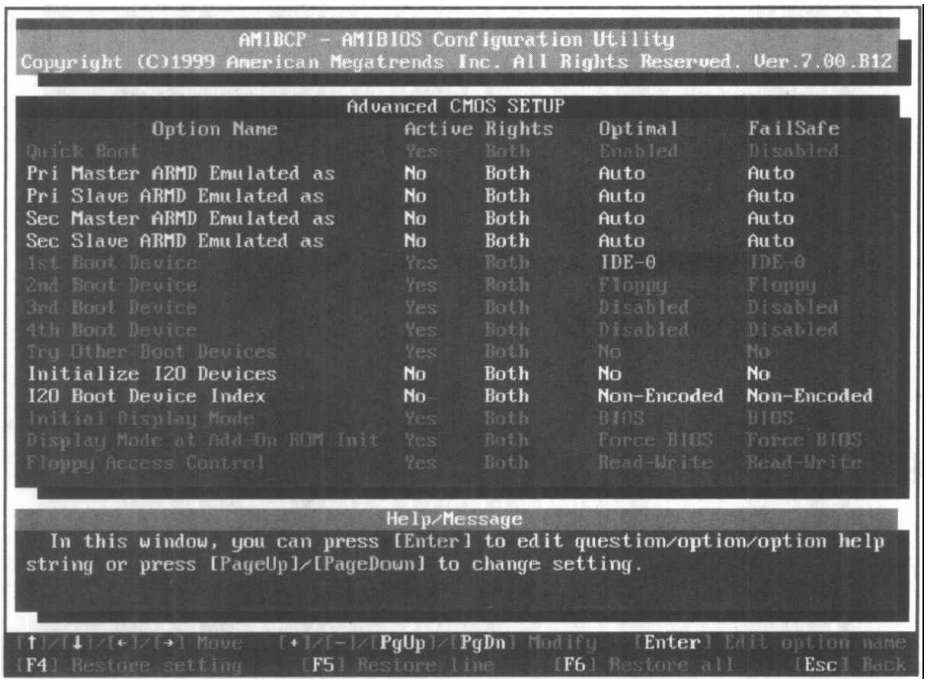


图 9-4　进阶 CMOS 设置画面

此时可对进阶 CMOS 设置画面中关于开机顺序、键盘按钮反应速率、快速开机模式、设置 Shadow RAM 内存区域、启用 PS/2 鼠标支持等项目预先设置，并且将组态值保存到 BIOS 文件，烧入 FlashEPROM 后随即生效。

③ **Chipset SETUP：** 芯片组参数设置。进入后出现如图 9-5 的设置画面。

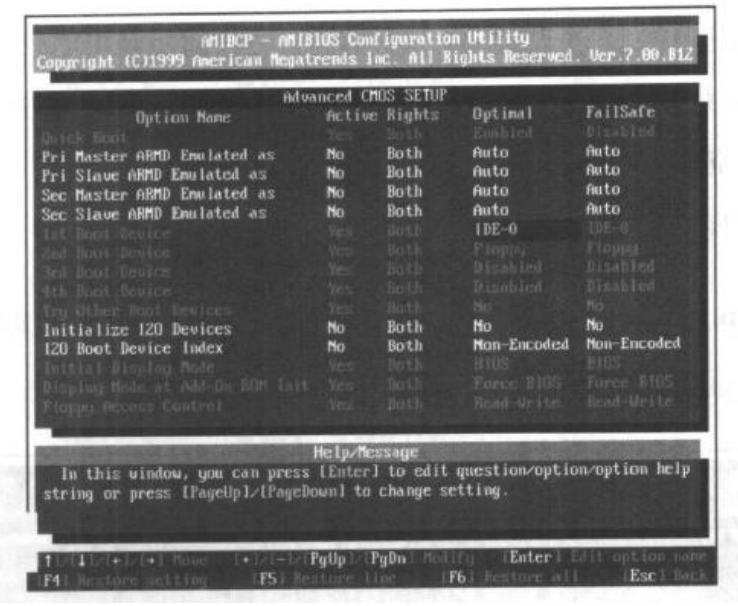


图 9-5　芯片参数组的设置画面

此时可针对芯片组参数设置画面，对影像共享内存大小、内存时序设置、硬件操作模式与启用 USB 鼠标与键盘等支持项目预先设置，组态值也保存到BIOS 文件中，日后一写入 FlashEPROM 后，开机随即以新的组态值加载而马上生效。

④ **Power Management SETUP**：电源管理设置。进入后出现如图 9-6 所示的设置画面。

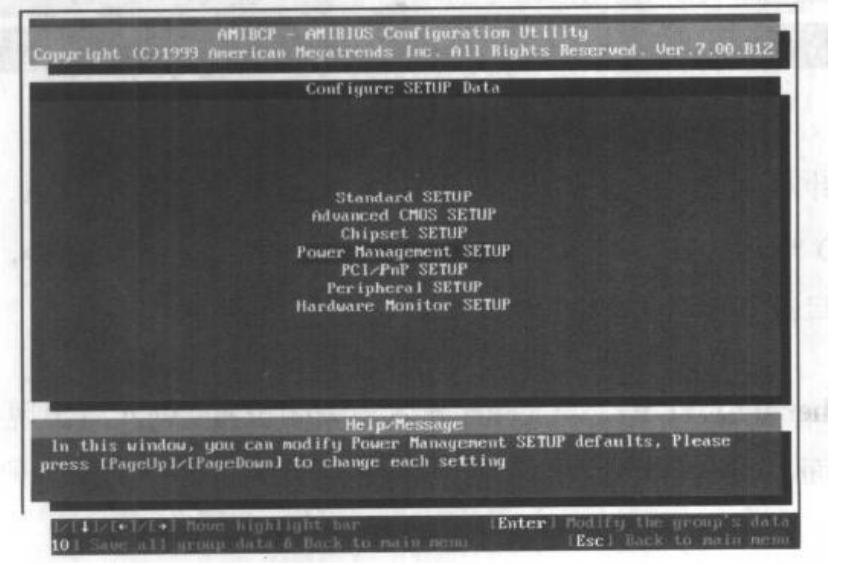


图 9-6　电源管理设置

此时可在电源管理设置下，对 ACPI，APM，ATX 软件电源开关设置，显示器、硬盘经过若干时间后关闭，以及进入节电模式后对哪些设备的运行做监控，一有动静立刻回复一般工作模式等项目，预先设置并存储组态到 BIOS 文件。

⑤ **PCI/PnP Setup**：PCI 外围与 PnP 设置。进入后出现如图 9-7 所示的设置画面。

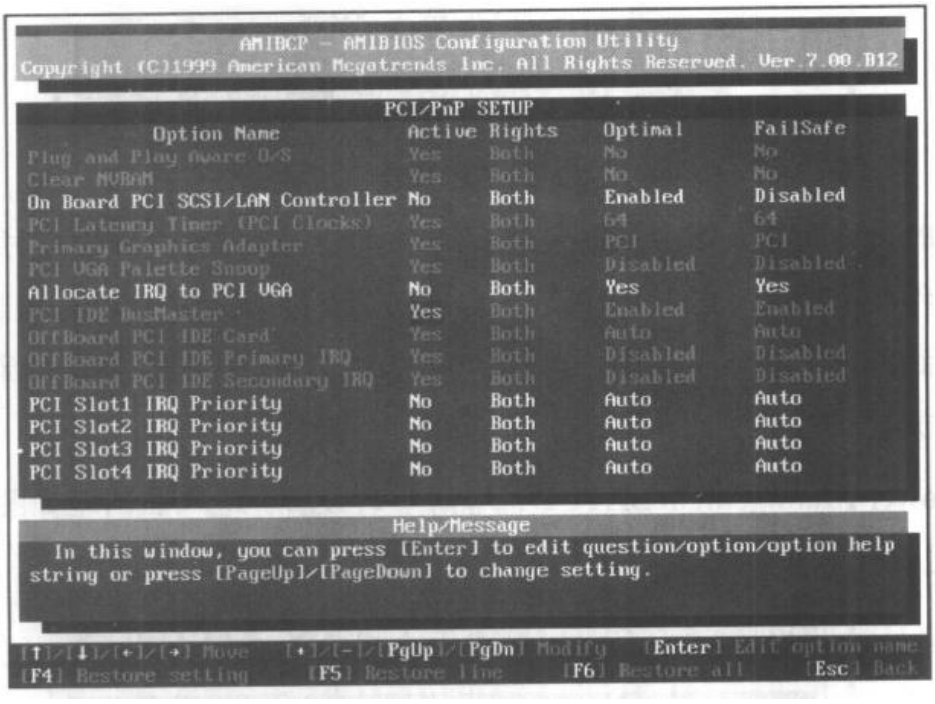


图 9-7　PCI 外围与 PnP 设置

这是针对 PnP 与 OS 的设置、NVRAM 清除以及各组 IDE、PCI 扩充槽的 I/O 端口、IRQ、DMA 的配置情况。这部分变动较少，也不需要先预写设置、保存组态到 BIOS 文件中。

⑥ **Peripheral SETUP**：外围装置开启 / 关闭设置。进入后出现如图9-8的设置画面。

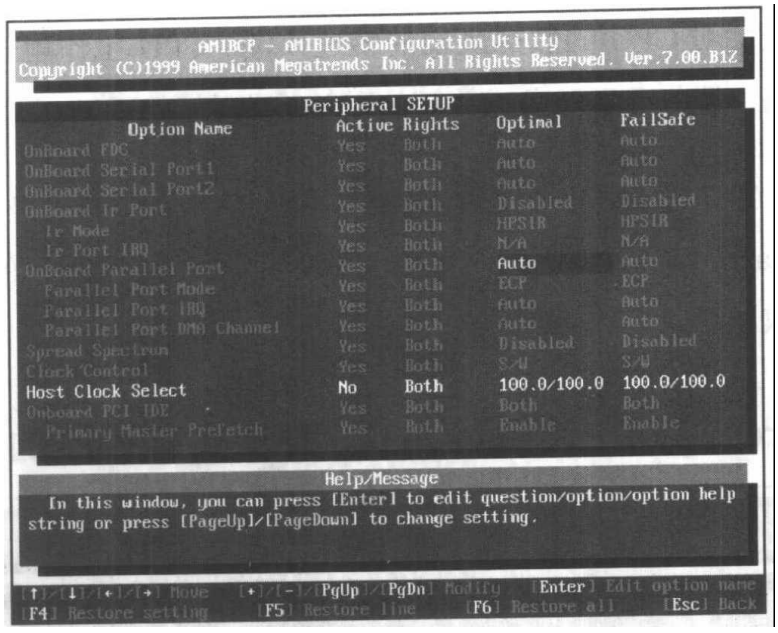


图 9-8　外围装置开启 / 关闭设置画面

这是针对外围装置开启 / 关闭与相关设置选项的画面。有时我们希望哪些装置默认值是开启的，I/O 端口、IRQ 与 DMA 号码是指向哪里，就可以充分运用这个设置画面，将所想要的默认值、开机值保存到BIOS ROM文件中，日后一写入 FlashEPROM，开机就是你想要的默认值。

⑦ **Hardware Monitor SETUP**：硬件状态监控设置，出现如图 9-9 所示的设置画面。

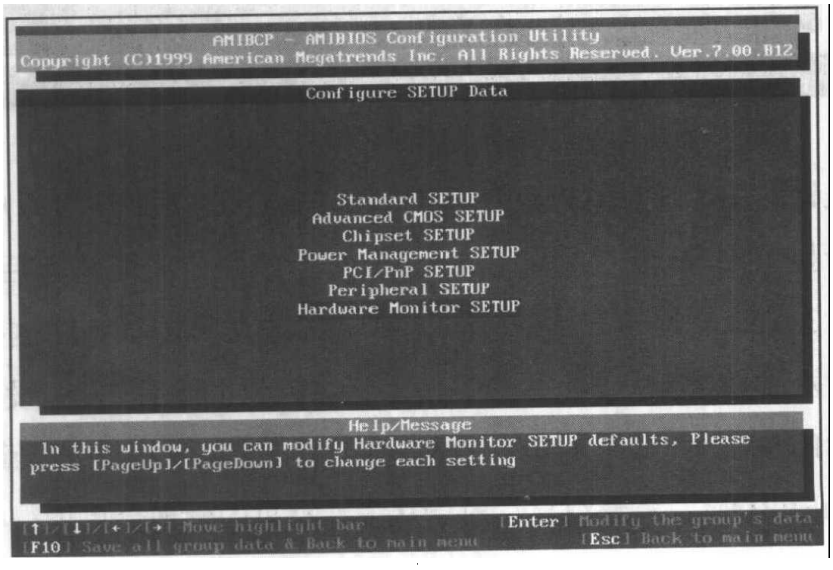


图 9-9　硬件状态监控设置画面

一般是针对 CPU 的散热风扇的转速侦测、CPU 工作电压、主板电压值的侦测，把你想要开启或关闭的默认值保存到 BIOS 文件。

其实在第 7 章 BIOS 画面、选项与默认值设计中，已经针对 Configure SETUP Data 这个项目，做了一次实际的说明。所以这个项目的实际应用，就不再重复叙述了。

Configure SETUP Client Data(设置接口程序的选项组态)

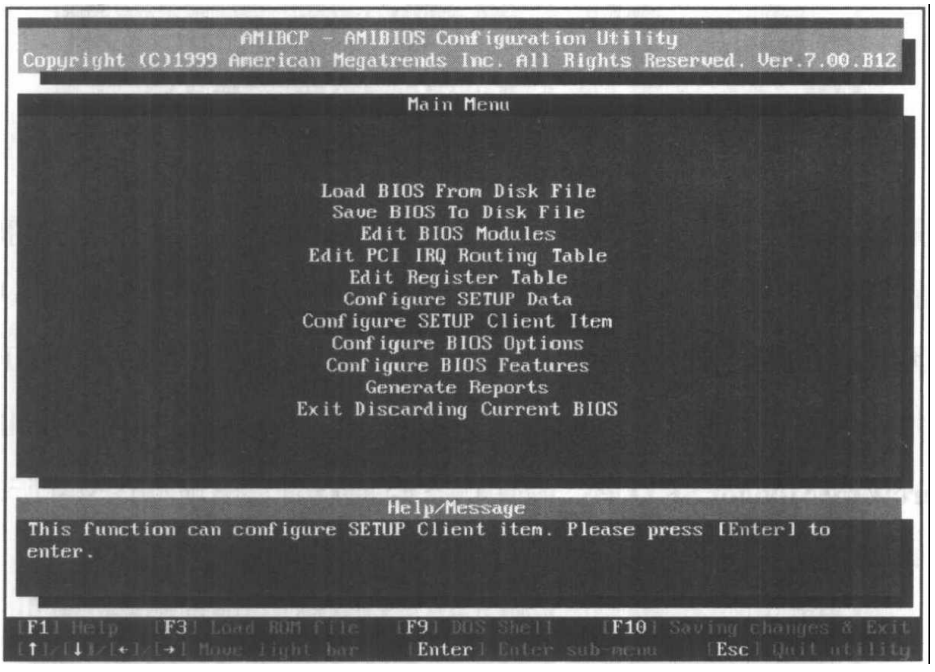


图 9-10 选中 Configure SETUP Client Item

这是针对 BIOS 设置画面出现时，哪些选项是：1.超级用户 / 一般用户都会显示出来，或者2.限定 supervisor 超级用户密码输入后才显示出来，或者3.Absent 隐藏这个选项。计算机维护人员可以善用这些选项，将一些涉及硬件参数，或者不希望使用者接触到的选项，预先设置组态并保存到 BIOS 文件(如图 9-11，直接把芯片组设置画面整个隐藏起来)，烧到 FlashEPROM，将使得日后对于众多计算机的管理会更加得心应手。

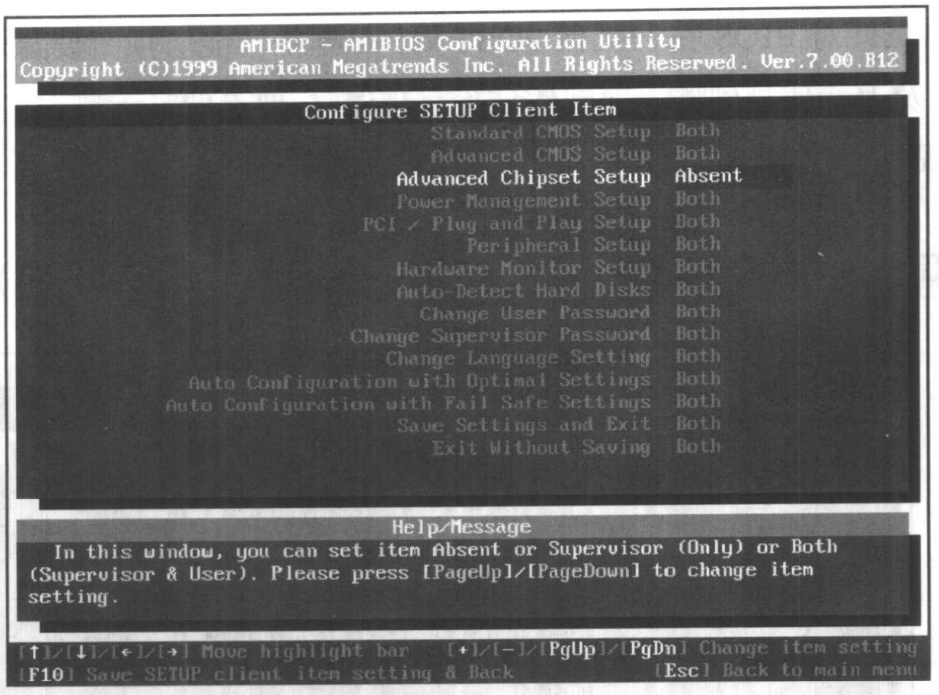


图 9-11　把芯片组参数设置画面隐藏起来

Configure BIOS Options(BIOS 显示 / 警示选项组态设置)

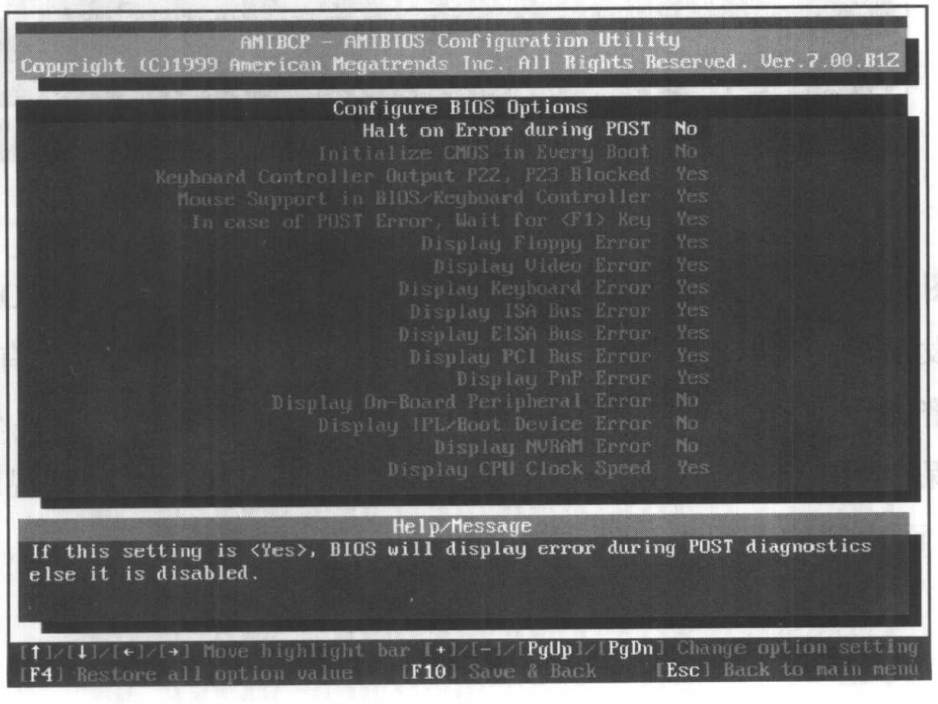


图 9-12　BIOS 显示 / 警示选项组态设置画面

这是针对 BIOS 在开机做 POST 时，有哪些功能 / 项目要显示(或不显示)，以及万一侦测到哪一项硬件装置有错误时，要不要显示错误信息以及应对的规则。

Configure BIOS Features(BIOS 特定内容组态设置)

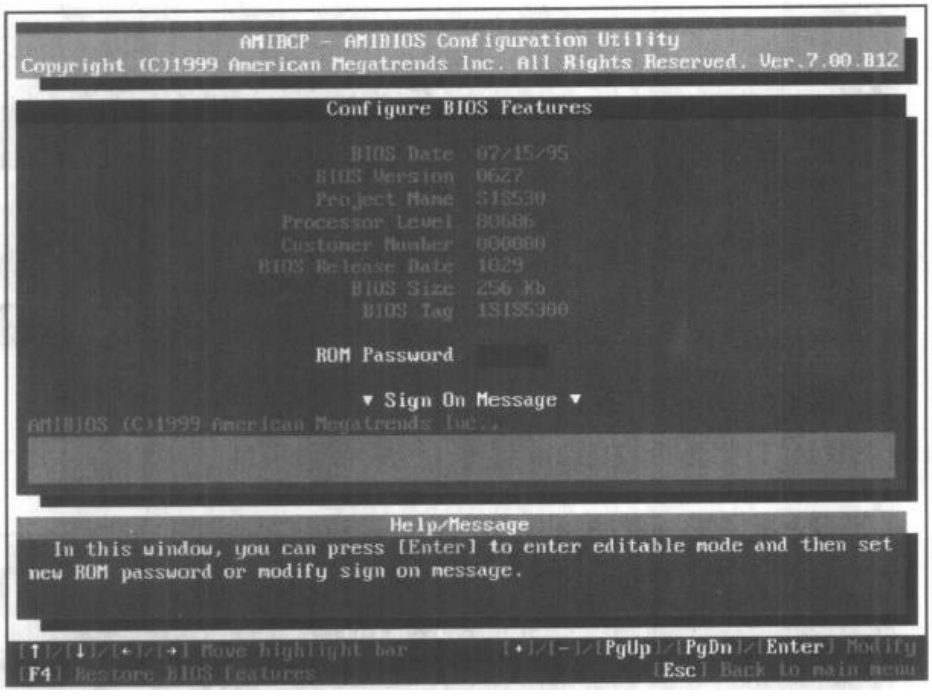


图 9-13　BIOS 特定内容的组态设置

图9-13是 BIOS 特定内容的组态设置。一般它只是显示 BIOS 纪念性的开发日期、大小、芯片组与处理器编号，能够修改的部分只有 ROM 的预设密码(如图9-14)，以及开机时的签名信息(如图9-15所示)。修改之后记得保存到BIOS 文件并烧到 FlashEPROM中，开机后就会显现新的签名信息，而内藏的密码也已经更改了！

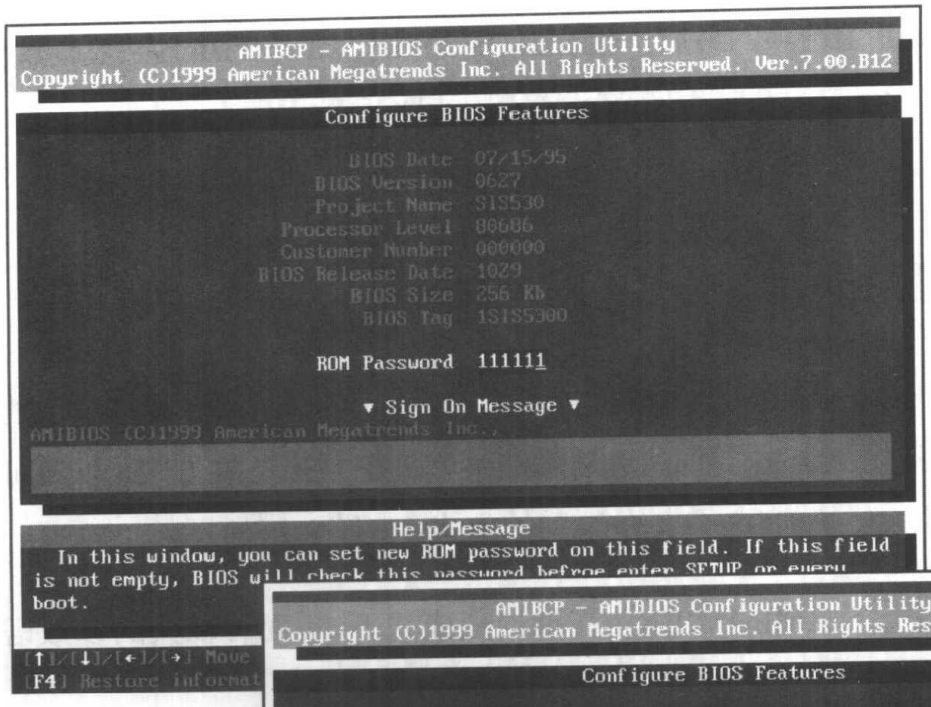


图 9-14 只能修改 ROM 默认密码

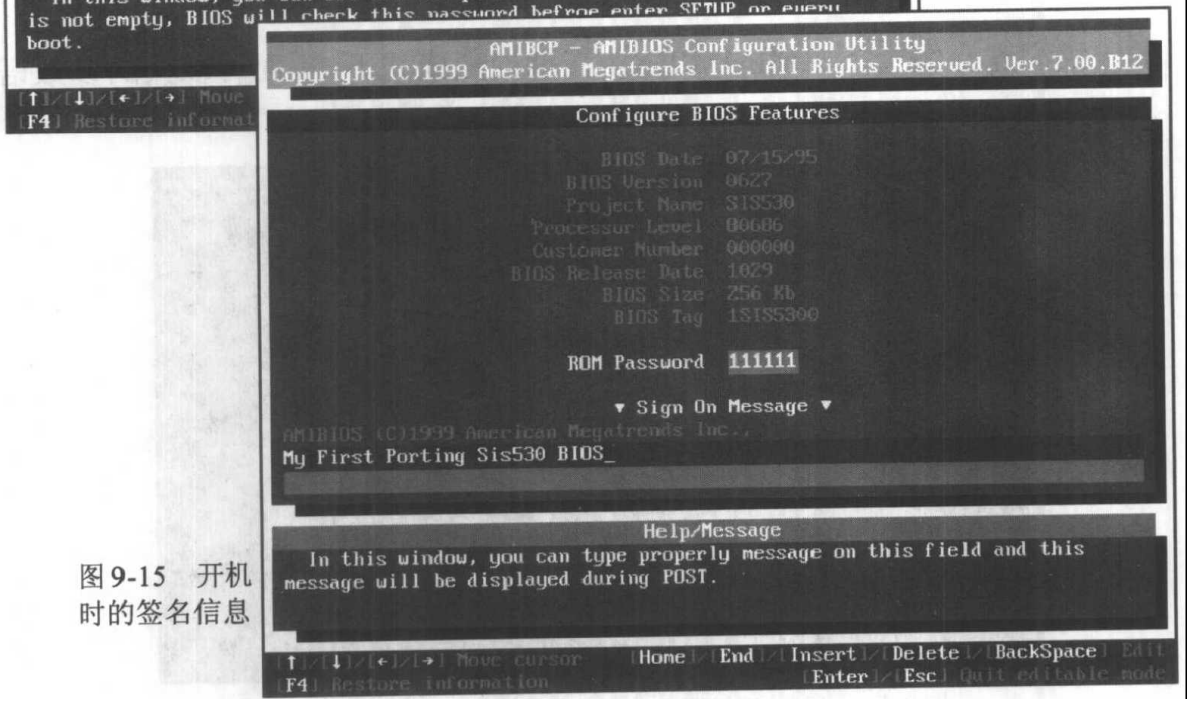


图 9-15 开机时的签名信息

Generate Report

倒数第二项 Generate Report 主选项，是针对读入的 BIOS 模块影像文件，做一份完整而详细的信息列表文件。读者可参考光盘所附 1SIS5300.RPT 这个文件内容。

至于AMIBCP主选项的最后一项Exit Discarding Current BIOS，意思就是放弃目前BIOS影像文件的一切组态设置，返回DOS或Windows环境。

9-2 AMIMM.EXE BIOS模块操作/修改程序

AMIMM.EXE（AMI ROM Module Manipulation Utility）上BIOS模块的操作/修改程序。它的操作界面，也是采取菜单/控制光条的方式，以及可搭配批处理文件的传统DOS命令参数列的方式，如图9-16所示。

我们先介绍一下AMIMM.EXE在批处理模式下，后面可以接的参数语法与用途。

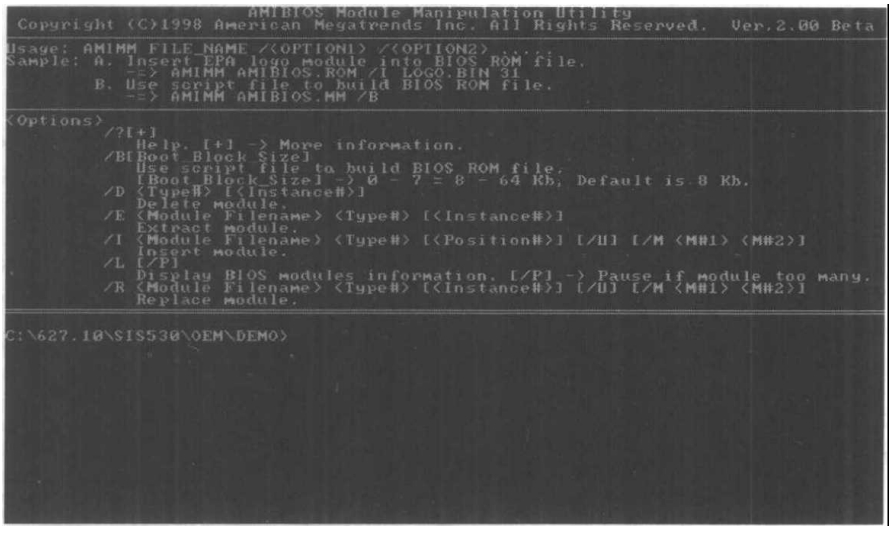


图9-16 AMIMM.EXE上BIOS模块在DOS下的操作界面

/B[Boot_Block_Size] 使用命令描述文件(扩展名为*.MM)来更改BIOS影像文件模块[Boot_Block_Size]，指定Boot Block的大小，其值从0～7，代表设置Boot Block从8KB～64KB，默认值为8KB(或视需要而定)

/D <Type#> [<Instance#>] 删除某个ID=Type#的模块为Type模块

/E <Module Filename> <Type#> [<Instance#>] 将某个ID=Type#的模块解压成指定的模块文件名

/I <Module Filename> <Type#> [<Position#>] [/U] [/M <M#1> <M#2>]
将某个模块文件插入指定位置 posotion#

/L
显示 BIOS 模块的详细信息

/P
当屏幕信息资料过多时，显示超过一个屏幕时先暂停一下，按一下按键再继续显示

/R <Module Filename> <Type#> [<Instance#>] [/U] [/M <M#1> <M#2>]
将某个文件中 Replace module.

如果是 AMIMM.EXE 后面跟了一个 biosfile.rom 的文件名，则直接进入图形界面操作模式，如下图 9-17 所示。

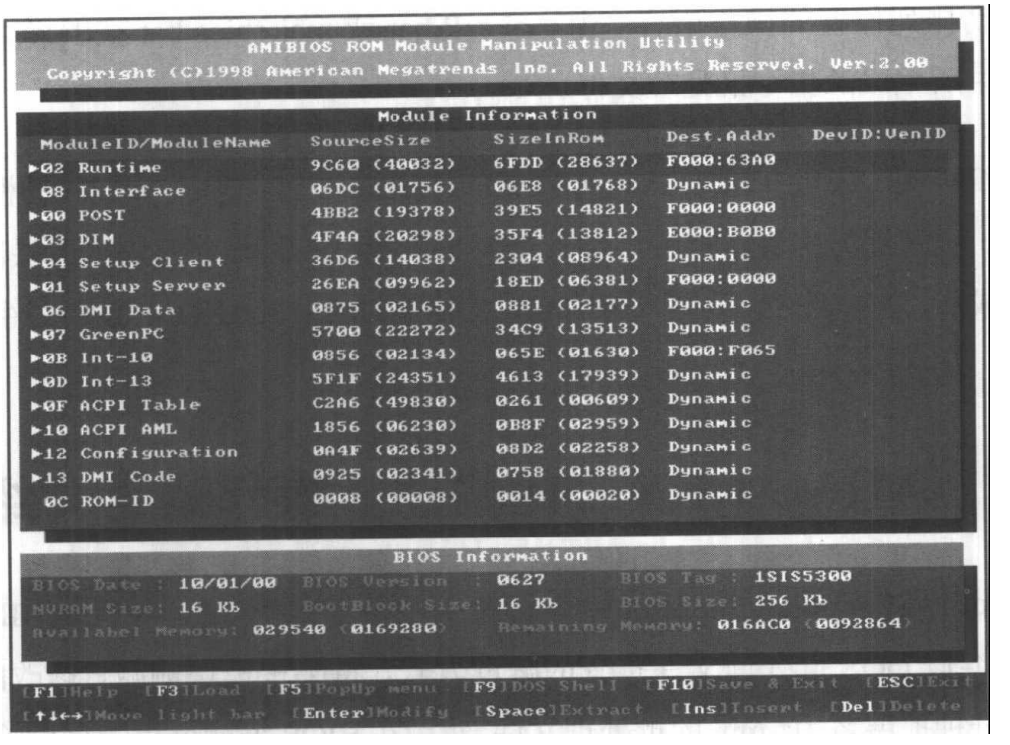


图 9-17　进入图形界面操作模式

此时你可以像 AMIBCP.EXE 的 Edit BIOS Module(编辑 BIOS 模块)的操作方式，通过[redacted]键移动控制光条，在可卷绕的列举模块中选择要改动的模块，按 Enter 就直接进入修改(可供修改的模块，请参考上图所示)，若要删除某一个模块，则按[redacted]键即可。

如果想要解压缩这个模块，并且另存为一个文件，则按下[redacted]之后，输入一个代表这个模块的文件名称就行了；若是要安插一个外加模块，则在按下Ins键之后，输入该模块的完整路径与文件名称就可以了。

基本上这部分跟AMIBCP.EXE的BIOS模块编辑画面、操作方式完全相同，这里就不再重复叙述。

最后，读者可以从所附光盘的\627.10\SIS530\ROM目录下，发现一个叫1SIS5300.MM的文件，它就是一个供AMIMM.EXE做内部模块删减、添加，以及更动排列位置的操作语言，由makeall.bat或makebios.bat在编译、链接的过程中所生成的指令批处理文件。

AMIMM.EXE会依照1SIS5300.MM的一道道命令叙述，对要加入的模块按顺序、位置逐一执行，把应该加入1SIS5300.ROM的每个模块排列妥当、设置完成后，才生成最后要写入FlashEPROM的1SIS5300.ROM影像文件。

9-3 AMIMM.EXE 的描述 / 操作语言介绍

Module Manipulation Script模块篡改描述语言文件，是一种让AMIMM.EXE以类似DOS批处理文件指令的解读方式，将某种模块删减、修改的方式写成一种扩展名为*.mm的批处理文件，然后在BIOS编译过后的最后阶段，调用AMIMM.EXE来逐一操作，完成最后一道模块排列、塞入的过程。

这套.MM的命令描述语法有点复杂，基本上每个命令都有一个RomVar(ROM影像文件ID，代表某个.ROM的影像文件)或ModVar(Module模块文件ID，代表某个正在操作的模块)，有时两个ID都需要指定。

接下来我们就按各命令的字母顺序作一下介绍。

ChangeDefaultDirectory("*Newdirectory*")

用途：切换目前的工作目录到 *Newdirectory*，路径与目录的字符串前后以“双引号”字符围起来。

举例：

```
ChangeDefaultDirectory("\627.10\SIS530\ROM")
```

切换工作目录到\627.10\SIS530\ROM 目录下。

ChangeModule(*ModVar*, "*modulefile*", ID=*Hexnum*, COMP=0|1, DESTSEG=*Hexnum*,DESTOFF=*Hexnum*)

ChangeModule(*ModVar*, "*modulefile*", ID=*Hexnum*,, COMP=0|1, DEVID=*Hexnum*,VENID=*Hexnum*)

用途：将目前工作的 ROM 影像文件的某个 *Hexnum* 模块，以新的 modulefile 模块读入并变更。

COMP=0|1 指定这个模块数据是否要压缩。如 COMP=0 不压缩，COMP=1 则先压缩再来做加入、排列的动作。

DESTSEG=*hexnum*、DESTOFF=*hexnum* 则是指定该模块预计要解压 / 执行的内存地址(DESTSEG)与内存偏移地址(DESTOFF)。例如设定 DESTSEG=F000, DESROFF=8000，则该模块会被解压到 F000:8000h 内存地址然后被调用、执行。

DEVID=*Hexnum*、VENID=*Hexnum* 表示被指定模块的 PCI DeviceID 与 Vendor ID。一般用于适合于 PCI 适配卡的 ROM BIOS 程序代码模块。

POS=nn，指定模块要插入、占据的第 nn 个位置(nn 从 0 起算)，若 POS=APPEND 则代表从最尾端插入。

以下是一个使用ChangeModule命令的例子。

```
ChangeModule(ModVar, "HF122.BIN", ID=04, COMP=1, DESTSEG=0000,
DESTOFF=0000);
Insert(RomVar, ModVar, POS=APPEND);
```

上述的意思是目前工作的 ROM 影像文件(代码为 RomVar)，将 HF122.BIN 模块读入，压缩后指定为 ID=04 的模块(Setup Client, 也就是设定接口程序)所指定的执行内存地址：偏移地址为 0000:0000(意义是动态解压执行，没有固定的内存地址)，然后插到该 RomVar 影像文件的后面(POS=APPEND 指定从后面添加)。

Close(*RomVar*|*ModVar*)

用途：将 RomVar 变量所代表的.ROM 影像文件(或 ModVar 变量代表的模块文件) 关闭，之后的操作命令对该影像文件或模块文件不再有影响。

Delete(*RomVar*, [ID=*Hexnum*, [INST=*Hexnum*]]|[POS=*Hexnum*])

用途：删除某个.ROM 影像文件内 ID 为 Hexnum 的模块。

INST=Hexnum，指定目前对应 RomVar 变量 ID 的副本，若 INST=0 或未指定，则改动结果直接影响 RomVar 所代表的 ROM 影像文件(或 ModVar 所代表的模块文件) 。

以下是一个运用 Delete 命令的例子。

```
delete(rom1,POS=3)      ←— 删除第三号位置的模块(从 0 起算，所以是第四个)
openrom(myrom,"1SIS5300.ROM")  ←— 开启1SIS5300.ROM这个影像文件,ID以myrom代表
delete(myrom,ID=36)     ←— 删除 ID 代码为 36h 的模块文件
```

Extract(*RomVar*, *ModVar*, [ID=*Hexnum*, [INST=*Hexnum*]||[POS=*Hexnum*]) Extract (*RomVar*, DEVID=*Hexnum*, VENID=*Hexnum*)

用途：将某个RomVar影像文件中的ID代码为Hexnum(或PCI DeviceID、VendorID号码)的模块解压出来，并且以ModVar为该模块的ID。

例如：

```
Extract(Rom1, suifile, ID=04)
```

← 目前Rom1的.ROM文件中ID代码为04的模块解压，以suifile这个变量代表这个解压模块的ID。

Insert (*RomVar*, *ModVar*, POS=[*Hexnum*|*APPEND*])

用途：将某个*ModVar*代表的模块插入到*RomVar*所代表的.ROM影像文件，插入的位置为POS=*Hexnum*或者APPEND(从尾端)。

例如：

```
insert(rom1, oemlogo, POS=APPEND)
```

← 将oemlogo代码所代表的模块数据插入rom1这个.rom的影像文件数据，从最尾端插入。

Link(*RomVar*, *ModVar*, DEVID=*Hexnum*, VENID=*Hexnum*, [INST=*Hexnum*]||[POS=*Hexnum*])

Link(*RomVar*, *ModVar*, ID=*Hexnum*, INST=*Hexnum*)

Link(*RomVar*, *ModVar*, ID=*Hexnum*, POS=*Hexnum*)

用途：将*RomVar*影像文件具有的*ModVar*模块数据，与PCI DeviceID=*Hexnum*、VendorID=*Hexnum*的附加ROM模块数据连接。

例如：

```
link(myrom, modvar, DEVID=6306, VENID=1039)
```

将 DeviceID=6306h，VendorID=1039h 的 PCI 附加 ROM 监控程序数据，一并连接到目前 Modvar 模块代码所代表的模块数据。

- **NewRom(*RomVar*, "*romfile*", SIZE=*hexromsize*, "*initfile*", NVRAMSIZE=nvramsize[, "*verstr*"])**

  用途：新打开一个.ROM 文件。文件名为 *romfile*，长度为 *hexromsize* (十六进制)，NVRAM 大小为 *nvramsize*，后面 verstr 为 ROM 版本号。本书范例为“0627”。

  举例：

```
NewRom(rom256, "1sis5301.rom", SIZE=40000h, NVRAMSIZE=8000h, "0627")
```

  打开一个新的.ROM 文件，文件名为 1SIS5301.ROM，长度为 256KBytes (=40000h)，NVRAM 的大小为 32KB(8000h)，ROM 版本字符串为 0627。

- **OpenModule(*ModVar*, "*modulefile*", ID=*hexnum*, COMP=0|1, DESTSEG=*Hexnum*, DESTOFF=*Hexnum*)**

  用途：打开一个文件名为 *modulefile* 的模块文件，指定 ModVar 为它的操作代码，以及它的模块代码 (ID=*hexnum*)，是否压缩(COMP=0 或 1)，解压执行的内存段地址与偏移地址等等。

- **OpenROM(*RomVar*, "*romfile*")**

  用途：打开一个文件名为 *romfile* 的.ROM 影像文件，并且以 RomVar 变

量作为代表这个.ROM 的操作代码。

Pause("*PauseMessage*")

用途：显示 *PauseMessage* 字符串信息并暂停，等待使用者按任何键后继续。

Play("*CommandLine*")

用途：执行一个特定的 DOS 命令。

例如：

```
Play("copy ...\..\rom\1sis5300.mm")   ← 将...\..\rom 目录下的 1SIS5300.MM 文件
                                        拷贝到目前的计划目录。
```

REM(*RemStatement1*) *RemStatement2*

用途：注释。*RemStatement1*，*RemStatement2* 均为注释的文字叙述。

Replace(*RomVar*, *ModVar*, [ID=*Hexnum*, [INST=*Hexnum*]|[POS=*Hexnum*])

用途：以新打开的 *ModVar* 模块 ID(所代表的模块文件)中 ID=*Hexnum* 的模块数据，以 INST 或指定模块顺序位置(POS=Hexnum)的方式，替换原先已打开的 *RomVar* 变量(所代表的 ROM 影像文件)。

例如：

```
OpenRom(myrom,"1SIS5300.ROM")   ← 将 1SIS5300.ROM 打开,指定为 myrom 这个变量
OpenModule(suifile, "win29.bin", ID=04, COMP=0, DESTSEG=0000,
DESTOFF=0000) ← 读入 win29.bin 这个设置接口程序，并指定它的 ID 为 suifile
                模块 ID 码为 04，采用动态执行

replace(myrom, suifile, ID=04)   ← 将 myrom 影像文件中的 ID=04 模块资料以 suifile 模
                                   块数据取代。
```

- **SaveAs(*RomVar*|*ModVar*,"*savename*")**

  用途：将 *RomVar* 或 *ModVar* 所代表的 ROM 文件或模块文件，以 savename 的文件名存储起来。

- **SaveInfoStructs(*RomVar*|*ModVar*,"*savename*")**

  用途：将 *RomVar* 或 *ModVar* 所代表的 ROM 文件或模块文件的数据结构，以 savename 的文件名存储起来。

以上介绍了共 16 个模块操作命令，像 AMIMM.EXE，以及前面介绍过的 AMIBCP.EXE，其实都能支持并解释*.MM 描述语言文件，帮你做批处理的模块增减、修改的动作。不过在实际的应用上，由于 AMI 所附带的 Makebios.bat 与 Makeall.bat 批处理文件，都已经把整体化编译、链接与移植(porting)过程都自动化了，由笔者查寻做出的*.MM(例如本范例 1SIS5300.MM)命令描述文件，内容中用到的命令其实不多，并没有把 16 道指令全用上。

事实上一般主板的 BIOS 的 RD 人员、维护人员，很少有机会要自己编写一个.MM 的描述文件来做模块篡改、调配的工作，一般都是使用操作界面较为简易的 AMIBCP.EXE 来完成。因此以上的.MM 描述语言文件，仅作为有兴趣对 AMIBIOS 编译、链接、移植过程作深入研究的读者另一项重要的技术参考文献。

9-4 RU.EXE 硬件系统寄存器检视程序

RU.EXE 是一套采用下拉式菜单，搭配窗口化操作界面的工具程序(如图9-18所示)。这在前面的几个章节中我们已经大致介绍到，RU.EXE 的功能主要在于可以协助 BIOS 研发与维护工程师，是进行寄存器填值检测、IRQ Routing 、内存、CMOS 内存与 Super I/O 查询的有力工具。

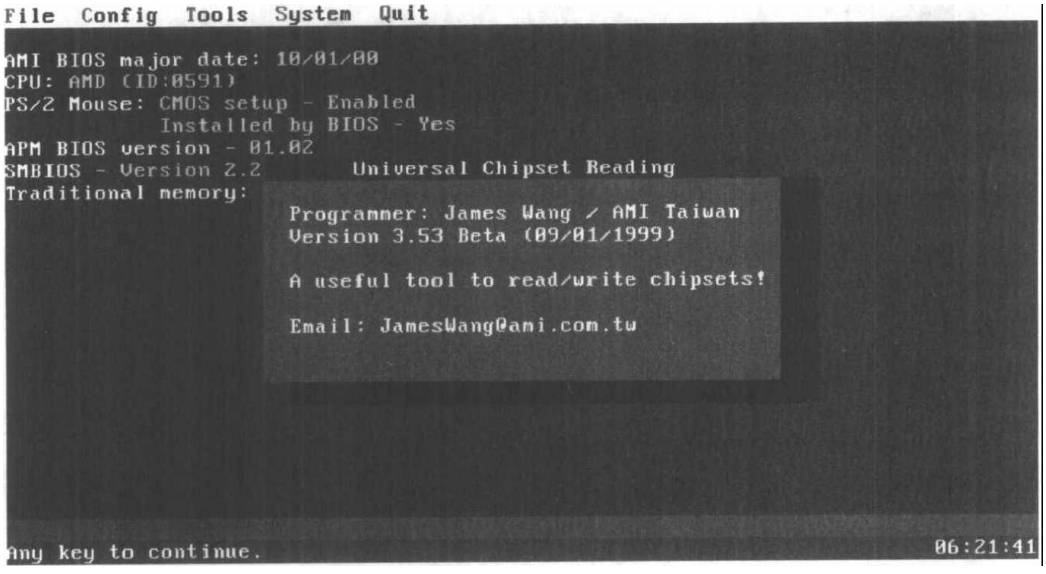


图 9-18　RU.EXE 的操作界面

如果你的主板不是使用 AMI BIOS，则会出现警告的信息，事后也会每隔一段时间蹦出类似的警告画面，但即使在非 AMI BIOS 的计算机下，一样能够执行无误，所以我们不必理会。

按下任何键后，进入一种十六进制数字的列举 / 编辑模式(如图9-19)，这种操作模式经常在许多类似扇区编辑的工具程序中看到，操作方法是按 [redacted]、[redacted]来选择要列举 / 编辑的 PCI 外围装置，用[redacted]、[redacted]、[redacted]、[redacted]方向键移动到所需要的数据栏位。

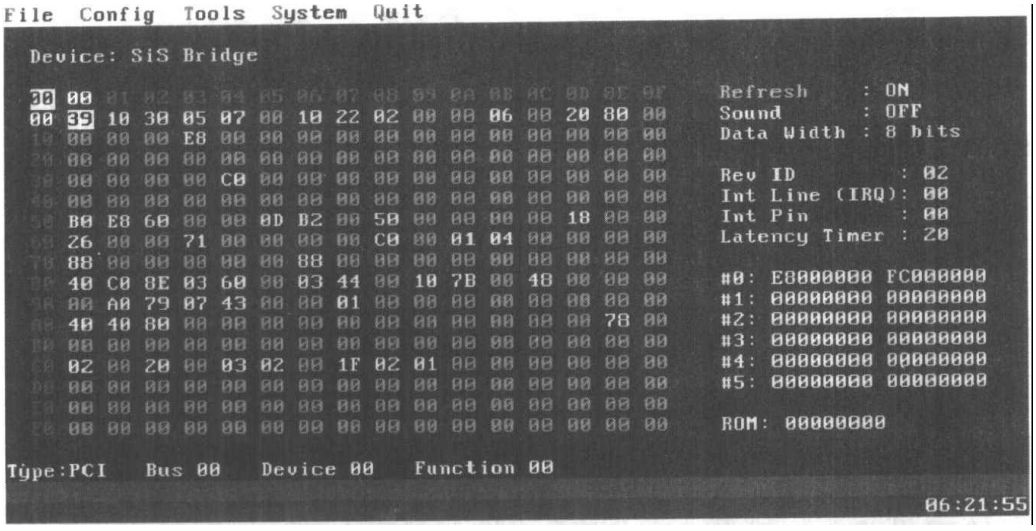


图9-19 进入16进位数字的列举/编辑模式

当你移动到某个具有代表性的偏移值字段时，最底下就会出现该字段的定义(如图9-20)。读者可以发现RU.EXE在16进制数字字段中，以黄色字体标示的数字字段，是建议可以修改的部分。

除非读者对PCI组态寄存器十分了解，也熟悉这个PCI装置的特性，要不然我们建议只在黄色的字段做改动，如果线上修改黄色字段以外的区域，可能会有死机或其他不可预料的后果。

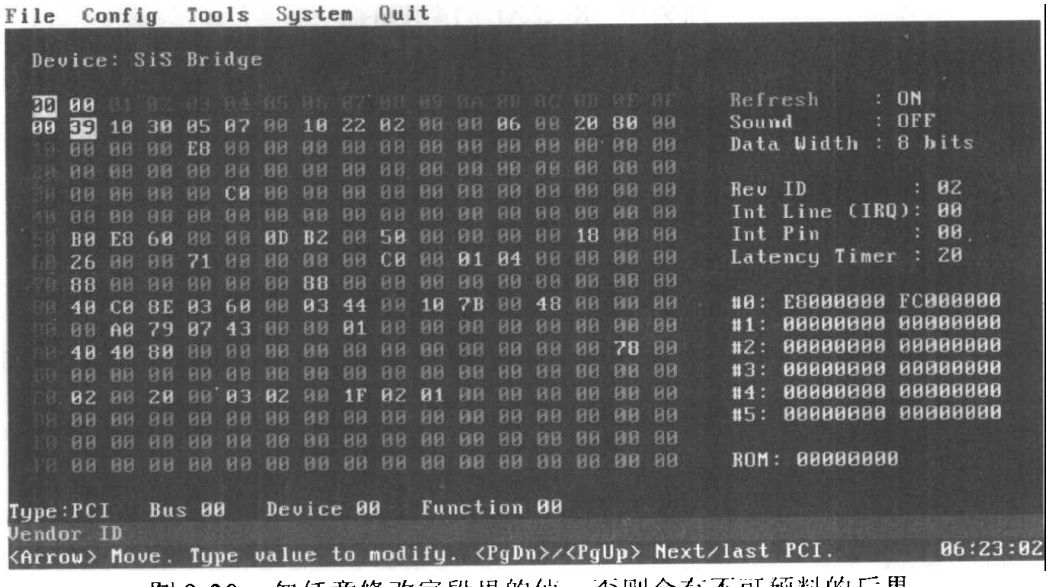


图9-20 勿任意修改字段里的值，否则会有不可预料的后果

右半部列举了一些可供参考的重要信息，如数据位宽度，装置的修正 ID 号码(Rev ID)，占用系统的 Int Line(IRQ)号码，硬件 Int Pin 的占用号码，Latency timer (起始延时计数值)，以及可能占用的内存位置或 I/O 端口地址的列举(#0～#5 字段，供 BIOS 与 Plug& Play 驱动程序侦测、辨识之用)等。

RU.EXE 的主菜单从左上角开始，依次是 File(文件功能)、Config(组态设置)、Tools(辅助工具)、System(系统功能)与 Quit(离开返回 DOS/Windows)，启用菜单的方式是按住 Alt 键的同时，再按住要选择的主菜单的第一个红色字母，例如要按出 Config 的选项，只要按 Alt + C 键就可以了。

以上五个主选项，除了 Quit 之外，其他主选项底下还有许多子选项，接下来我们陆续介绍这些子选项，并且选择较常用的子选项做举例说明。

一、File (文件功能)

含有 1.Load(加载硬件组态设置文件)，如图 9-21 所示。2.Save(将目前硬件组态存盘)。3.Compare(将目前硬件组态跟指定的硬件组态文件作比较)。

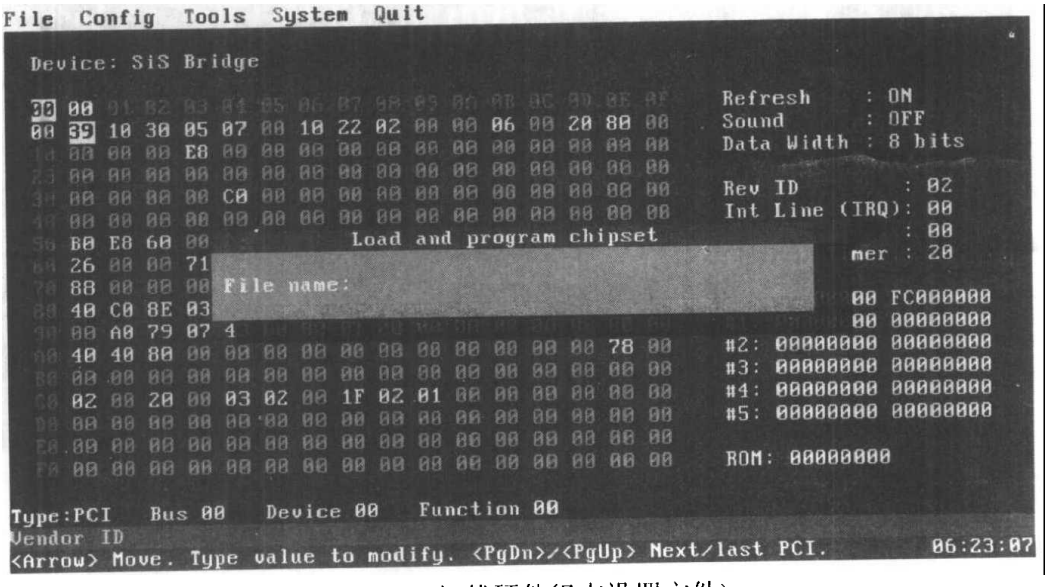


图 9-21 Load (加载硬件组态设置文件)

以上这些选项，有助于目前的硬件组态参数的取得与存盘，或者从其他相同结构的芯片组取得硬件参数文件来作对比，也便利于主板硬件参数的测试、读写以及 BIOS 移植。

二、Config(组态设置)

有 1.PCI (PCI 设备组态)；2.ISA (ISA 设备组态)；3.ISA IO (ISA IO 装置组态)，含 CMOS、SuperI/O 芯片的组态设置(图 9-22)；4.IO Space (IO 口读写)；5.IDE Identify(IDE 硬盘参数检测)，如图 9-23 所示；6.Memory (Flat) 指定内存地址并读写修改(图 9-24，9-25 所示)；7.Open Config(开启 SuperI/O 芯片的特殊组态模式)；Close Config(关掉 SuperI/O 的特殊组态模式)。

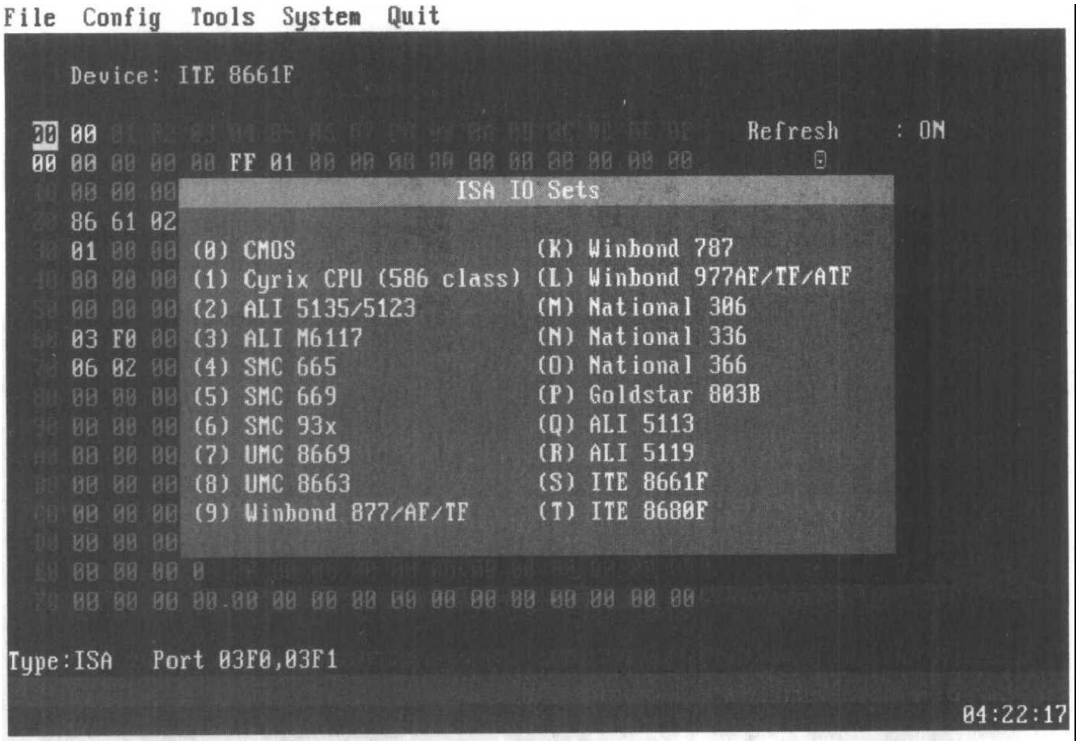


图 9-22 含 CMOS、SuperI/O 芯片的组态设置

```
File  Config  Tools  System  Quit

  IDE HDD: IBM-DHEA-34330

  30 00                                                    Refresh     : ON
  00 5A 04 D0 20 00 00 10 00 00 00 00 00 3F 00 00 00       Sound       : OFF
     00 00 00 00 20 20 20 20 20 20 20 20 4A 20 30 33       Data Width : 8 bits
     33 4A 34 30 32 37 35 32 03 00 B8 03 1C 00 45 48
     4F 35 34 41 46 30 42 49 2D 4D 48 44 41 45 33 2D       Cylinders: 8400
     33 34 30 33 20 20 20 20 20 20 20 20 20 20 20 20       Heads: 16
     20 20 20 20 20 20 20 20 20 20 20 20 20 20 10 00       Sectors: 63
     00 00 00 0F 00 00 00 02 00 02 07 00 D0 20 10 00
     3F 00 00 33 81 00 10 01 00 33 81 00 07 00 07 00     PIO Mode: 4
     03 00 78 00 78 00 F0 00 78 00 00 00 00 00 00 00       Multiword DMA: 2
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00       Selected MDMA: None
     0E 00 06 00 09 00 00 40 00 00 00 00 00 00 00 00
     07 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00       Ultra DMA mode: 2
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00       Selected UDMA : 2
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Type:IDE    Offset: 000
```

图9-23 IDE硬盘参数侦测

```
File  Config  Tools  System  Quit
<Arrow> to
```

```
  30 00                                                    Refresh     : ON
  00 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     31 53 49 53 35 33 30 30 00 00 00 00 00 00 00 00       1SIS5300
     EA 5B E0 00 F0 31 30 2F 30 35 2F 30 30 00 FC 00       Ω[α ≡10/05/00 ⁿ

Type:Memory    Address 000B8000
```

图9-24 Memory指定内存地址并读写修改

```
File  Config  Tools  System  Quit                                  04:21:18
```

```
  30 00                                                    Refresh     : ON
  00 46 FC 69 F0 6C F0 65 F0 20 F0 20 F0 43 FC 6F F0       Fⁿiπlπeπ π πCⁿoπ
     6E F0 66 F0 69 F0 67 F0 20 F0 20 F0 54 FC 6F F0       nπfπiπgπ π πTⁿoπ
     6F F0 6C F0 73 F0 20 F0 20 F0 53 FC 79 F0 73 F0       oπlπsπ π πSⁿyπsπ
     74 F0 65 F0 6D F0 20 F0 20 F0 51 FC 75 F0 69 F0       tπeπmπ π πQⁿuπiπ
     74 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20    tπ π π π π π π π
     20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20     π π π π π π π π
     20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20     π π π π π π π π
     20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20     π π π π π π π π
     20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20     π π π π π π π π
     20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20 F0 20     π π π π π π π π
     20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20    ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲
     20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20    ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲
     20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20    ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲
     20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20    ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲
     20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20    ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲
     20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20 1E 20    ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲ ▲

Type:Memory    Address 000B8000

<Arrow> Move. Type value to modify. <PgDn>/<PgUp> +/- 100h bytes        04:21:32
```

图9-25 修改后的画面

以上各子选项中，第3项是ISA IO组态，选CMOS则会列出0070h、0071h以及对应的128个CMOS内存值(如图9-26)，你会发现有几个字段数据内容会动态跟着变动(因为那就是CMOS记录系统时间的秒数、分钟数的BCD值)。

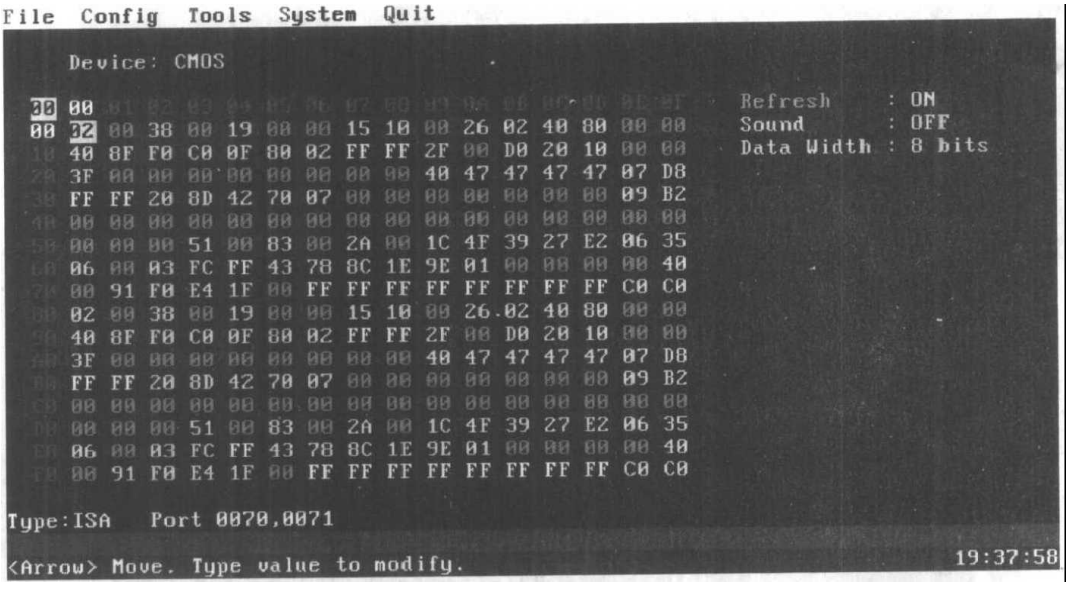


图9-26 选CMOS时0070h、0071h对应的CMOS内存值

至于IDE Identify(IDE硬盘参数侦测)，经笔者测试得出：必须在纯DOS环境下，而且不能挂任何的EMM386.EXE、QEMM386.SYS等内存管理程序，才能正确读取指定硬盘的参数(cylinder/head/sector)，以及厂牌字符串的信息(如图9-27)。

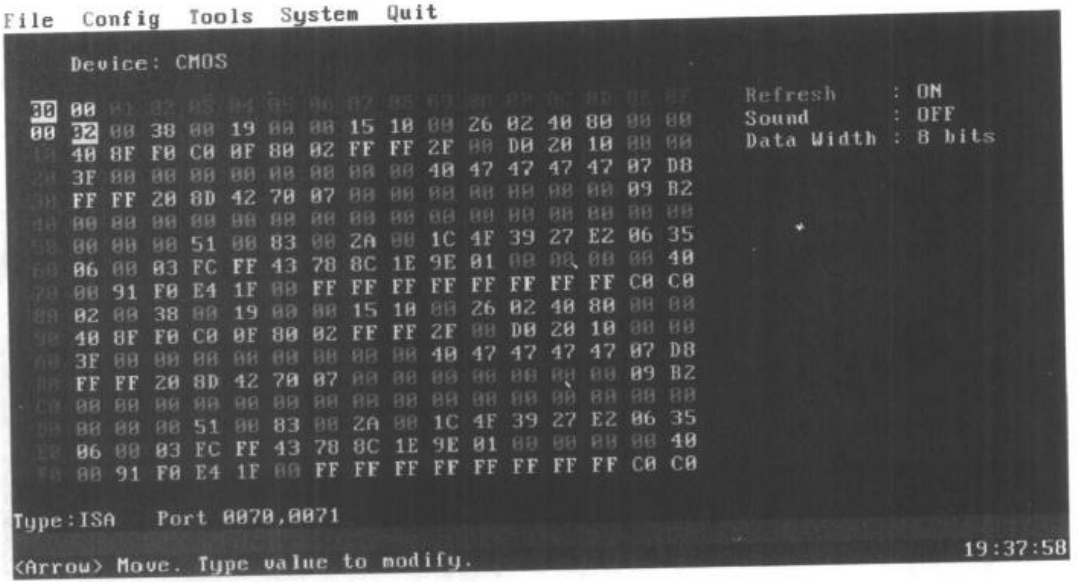


图 9-27 在纯 DOS 环境下，才能读取指定的硬盘参数

这时硬盘参数的 I/O 端口读写有着紧密的时序限制。在纯 DOS 模式下执行 RU.EXE，此时对 I/O 端口读写毫无限制，时序完全不会被耽搁，而 Windows 下进入所谓的 386 保护模式，也只有 ring0 阶层的驱动程序(.VXD)才能直接全速、毫无限制地动用 I/O；切换到 MS-DOS 模式时，许多 I/O 端口的读写是以虚拟化(或者说拦截后仿真)方式进行的，真正从该 I/O 端口读写的速度会变得很慢，因此便无法顺利工作。

另外像第 6、7 项，必须在选择一个正确的 Super I/O 芯片类型之后(如图 9-28 所示)，再选择第 6、7 项才会有意义。

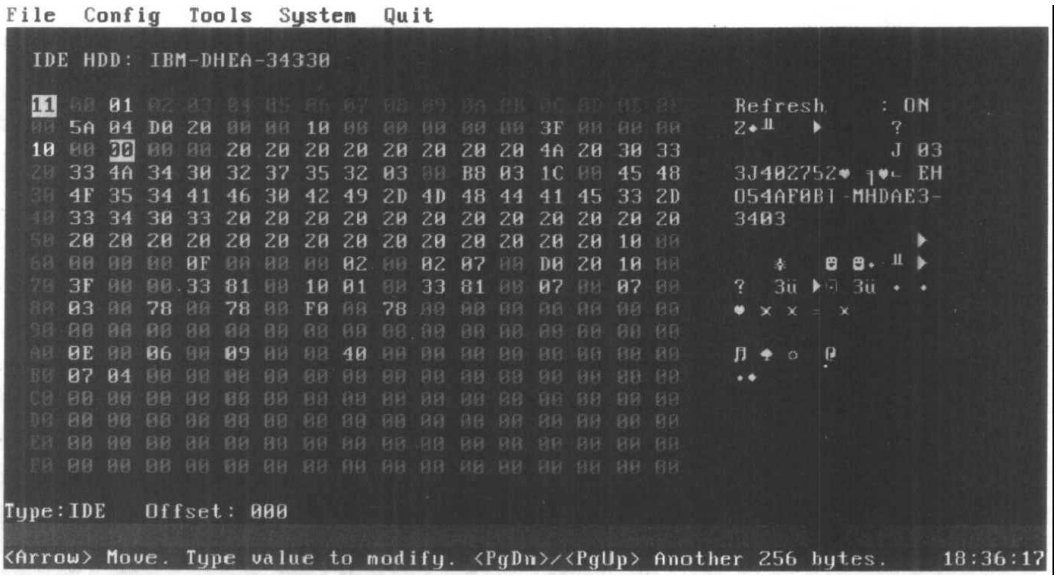


图 9-28 先选择一种正确的 Super I/O 芯片

以本书选用的ITE8661F Super I/O 芯片作为范例，则 RU.EXE 会列出该芯片的 Open Config 组态命令(如图9-29)，此时你可以记录下来另做他用(写入前面介绍的源代码)，或切换到其他 I/O 端口画面来检视 Super I/O 的状态。

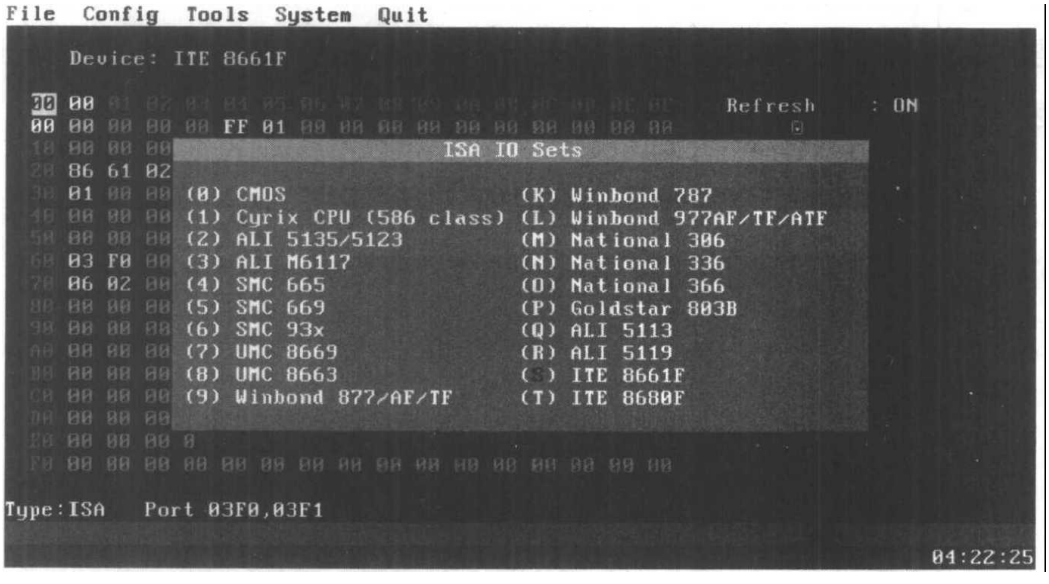


图 9-29 ITE8661F 芯片的 Open Config 组态命令

三、Tools(辅助工具)

① List PCI devices 列出了所有 PCI 装置，也可按 ■ 键，如图 9-30 所示。

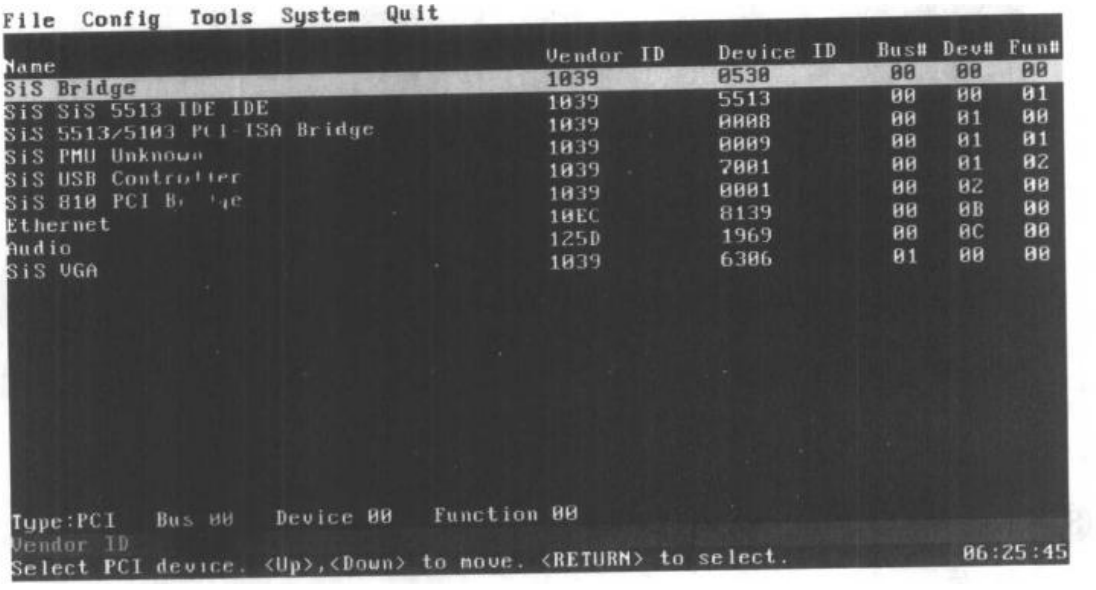


图 9-30 列出的所有 PIC 装置

② Toggle 8/16/32 bit 切换数据格式显示，或按 ■ 键可连续切换为 16 位(图 9-31）或 32 位显示（如图 9-32)。

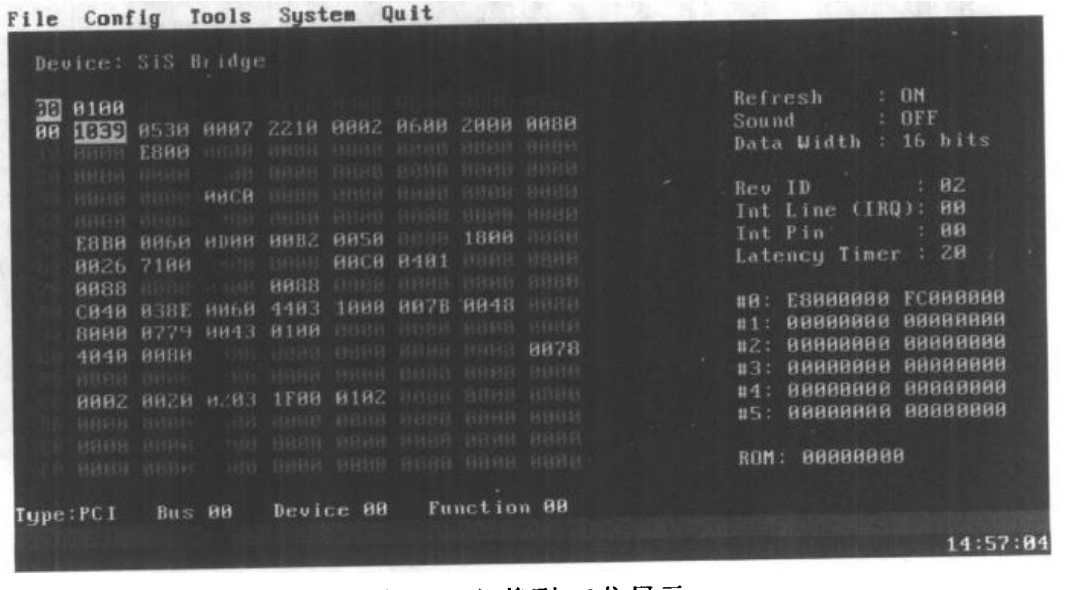


图 9-31 切换到 16 位显示

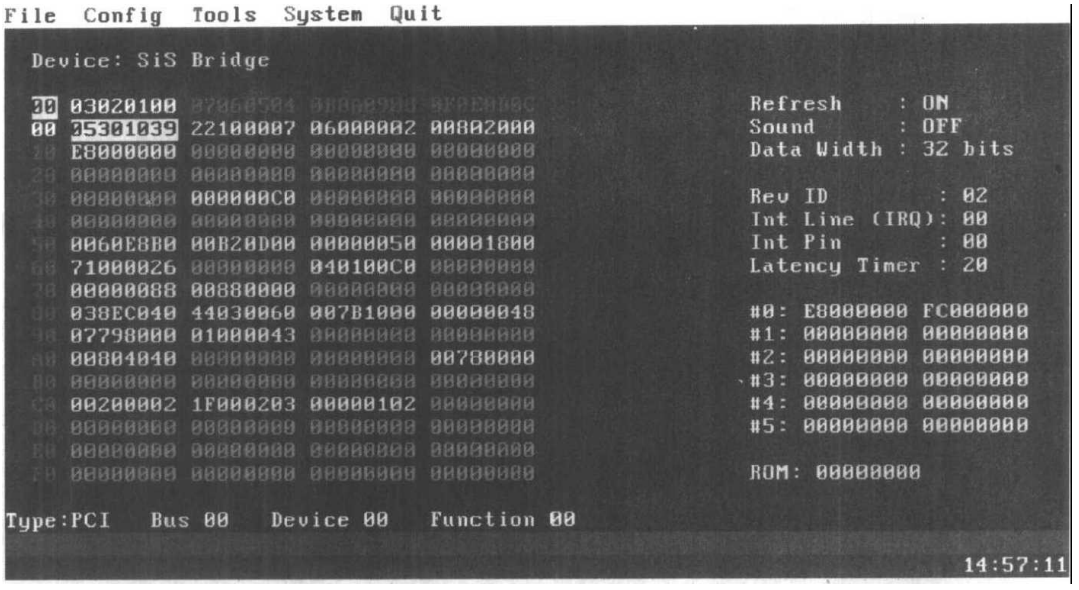


图 9-32　切换到 32 位显示

③ Toggle ASCII/Info 切换右半边为 ASCII 字符(如图9-33)或到原先的信息提示画面。

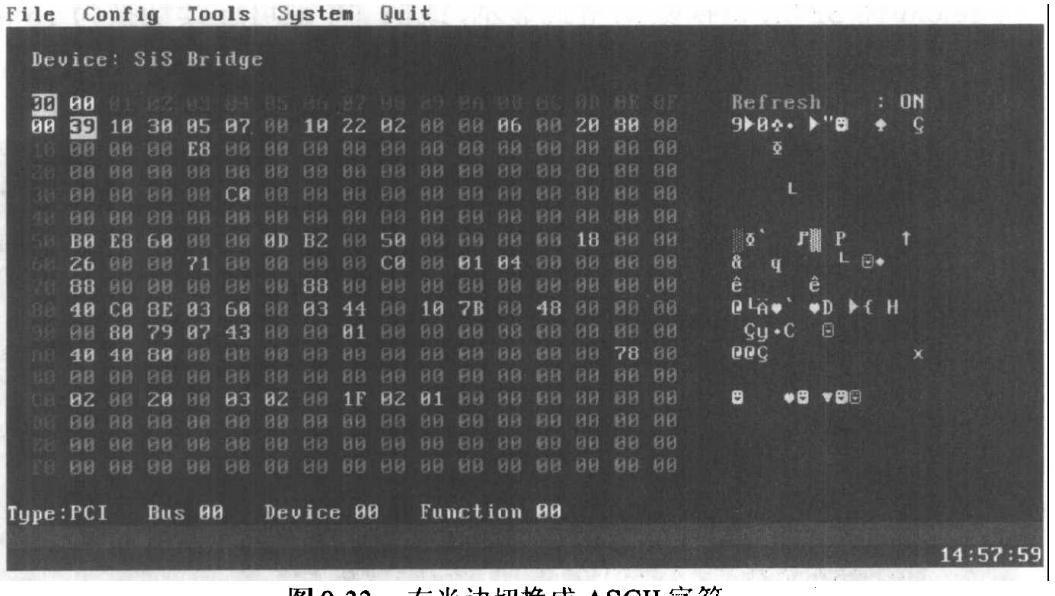


图 9-33　右半边切换成 ASCII 字符

④ System Info(列举系统信息)，也可以按Ctrl+F1键。

⑤ Help 显示求助画面，按F1效用相同。

以上这些选项，像第 1 项可以列出目前计算机平台(主板)上所安插的 PCI 适配卡，或内置的 PCI 芯片装置等详细信息，这个画面对以后做 PCI IRQ Routing 时非常有帮助，在稍后的移植实作篇中会再仔细地说明。

四、System(系统功能)

这个选单中有 1. Change Color(切换颜色)，按Alt+C键；2. Toggle Sound(切换静音 / 声音模式)；3. Reboot INT19(以 Int 19h 中断调用加载操作系统)；4. Turn off ATX Power(关掉 ATX 主板的电源)。这些功能都非常单纯，在此就不多加说明，留给各位 BIOS 系统研发者或读者自行研究吧！

9-5 其他研发工程师 / 高等玩家的工具程序

接下来我们依照文件字母的顺序，介绍这些研发工程师与高等玩家会使用到的工具程序。读者也许会发现，有些工具程序的功能会有重复或重叠的情形，这可能是因为程序开发时间不同，或者当时设计的任务需要所导致的必然结果。因此，有些比较老旧、功能可被取代的工具程序，或者 AMI 正在进行大刀阔斧、重新编写改变的工具程序，甚至是自己使用的工具程序，因为篇幅上的限制，我们在此就不做太多的说明。

AMIEMBED.EXE

AMIEMBED.EXE是针对一般以及特定ROM模块做监控程序嵌入(embeded)的工具程序，目前的版本号是v6.25t。在DOS提示符号下输入AMIEMBED，可以出现它的语法说明。

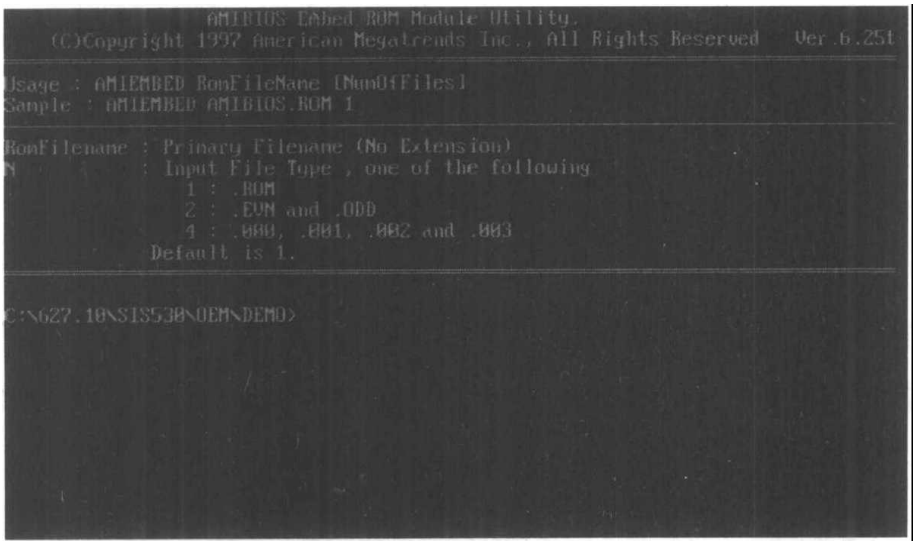


图9-34　在DOS环境下执行AMIEMBED

在早些时候有些EPROM、FlashEPROM芯片的数据宽度只有8位的年代，由于286AT所定义的AT Bus是16位的数据宽度，若要使用8位的EPROM/FlashEPROM芯片，一次需要用上两块，一块存放奇数字节(Odd Bye)，一块存放偶数字节(Even Byte)，若是搭配一些32位总线宽度的适配卡/外围装置，就得用上四块。

因此，AMIEMBED后面所接的RomFileName，可以是.ROM的形式(单颗/连续存放形式)，奇特的.EVN，.ODD(两块、奇数/偶数位各自存放)，以及.000，.001，.002，.003 （四块、四字节分离各自存放)的各种类型的ROM监控程序影像文件，可以针对不同需要将监控程序做奇、偶数字节分离存放/读写，或四组byte分离存放/读写。

至于 AMIEMBED.EXE 的使用方法，是以窗口卷绕的选项 / 模块方式呈现，可以用↑、↓键移动控制光条，到你所要选择的模块字段，如图 9-35 所示。此时就可以按Enter做修改，按下Insert 键指定要插入的模块，按Del删除所选择的模块。

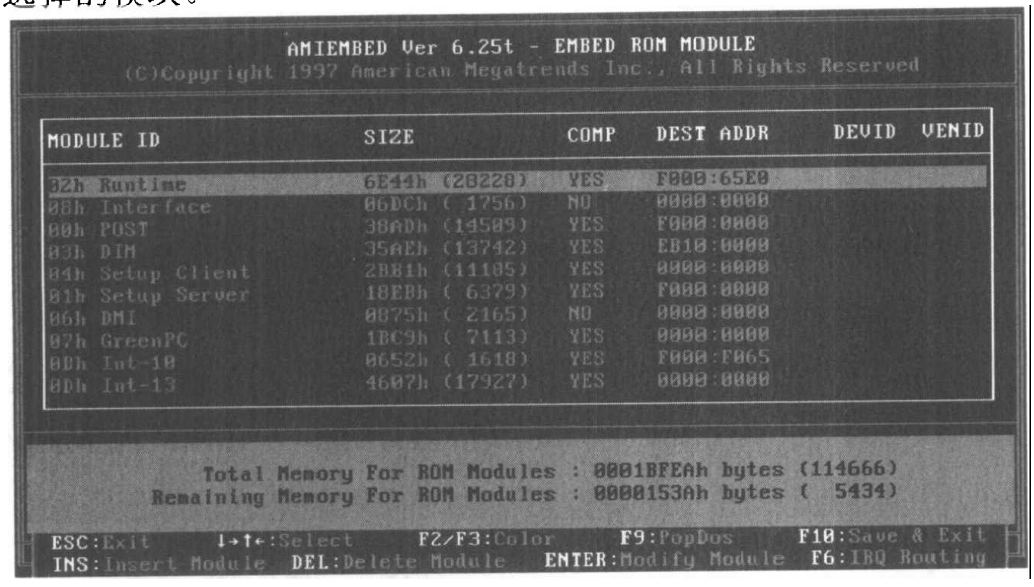


图 9-35　选择模块字段

另处，也可以按 F6 键来做关于 IRQ Routing(IRQ 绕线)的设置，如图 9-36 所示。

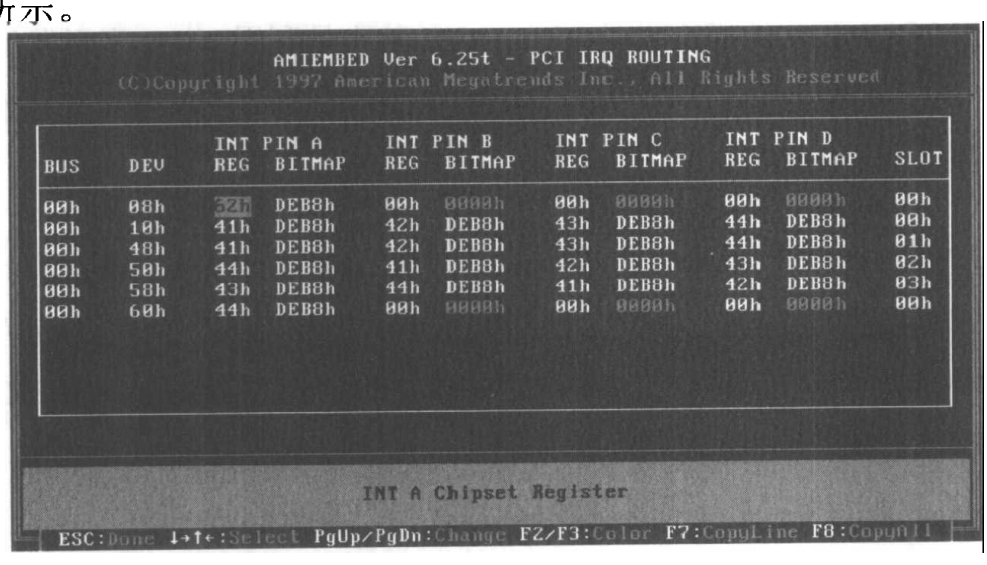


图 9-36　IRQ Routing 设置

修改、调整过的 BIOS 影像文件，记得按F10保存。

眼尖的读者可以会发现，AMIEMBED.EXE 的功能、画面，俨然就是最早期 AMIMM.EXE、AMIBCP.EXE 的精简版本，也许是AMI 工程师在研发 AMIEMBED.EXE 的过程中，不断地添加、扩充新的功能。

不过 AMIEMBED.EXE 毕竟是比较老的工具程序，笔者在写这个工具程序介绍时，发现它最大只能读取 128KB 的 BIOS 影像文件，而本书要 porting 的 256KB Sis530 芯片组 BIOS 还无法被读出来，会出现 Current Disk is Full !的信息。为了介绍这套软件，笔者是以删减模块的方式，另外造一个容量刚好为 128KB 的文件，才得以取得必要的图形来做说明。

所以，AMIEMBED 这个工具程序，读者应该不太会用到，而它的功能是下面要介绍到的 AMIMM.EXE，以及前面介绍过的 AMIBCP.EXE 都可以做得到的。

AMIOLG.EXE

原本 BIOS 开机时右上方所显现的绿色EPA 能源之星图形，是 AMI 预设的 OEM Logo，它的模块号是 31h ，AMI 其实是利用 VGA 字形发生器的技术，利用重新定义特定字符的字形数据(位图形)的方式，以显示文字的方式来达到显现图形的效果。

AMIOLG.EXE 是AMI 公司提供给OEM 厂商的小Logo 商标画面的修改程序。它可以将一个事先用影像软件绘制好的.BMP 图形文件，转换成能被 AMIBIOS 所识别的字形数据模块，然后只要把这个模块通过 AMIMM.EXE 或 AMIBCP.EXE 加入到 BIOS 监控程序中，烧到主板的 FlashEPROM，下次开机时，就会看到 OEM 厂商或读者自己制作的 OEM Logo 啦!

我们先来介绍 AMIOLG 的使用语法，如图 9-37 所示。

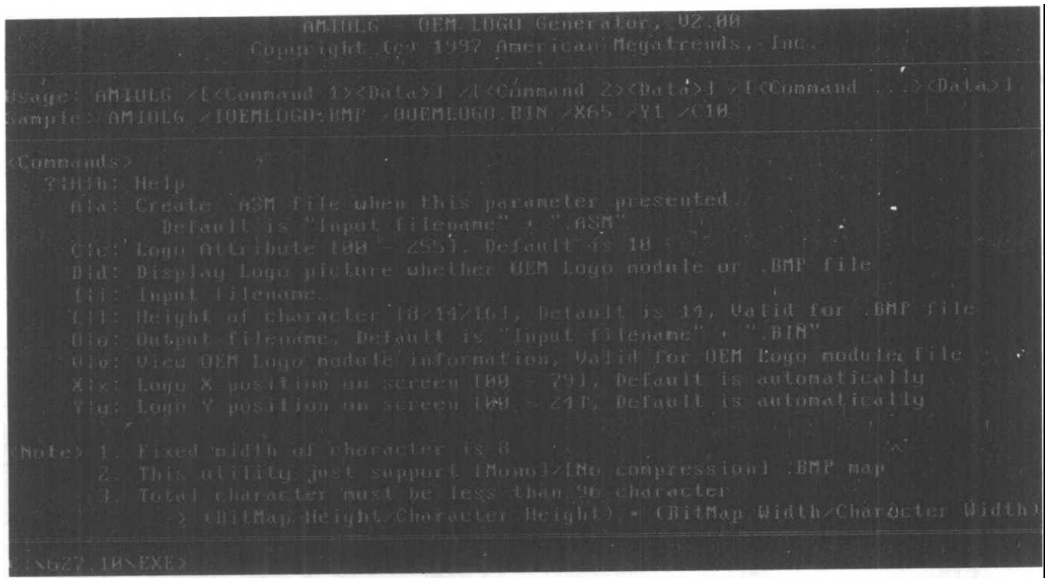


图 9-37　在 DOS 环境下执行 AMIOLG.EXE

?、H 或 h：显示所有参数的命令与语法。

A 或 a：生成指定文件名后扩展名为.ASM 的汇编语言源文件。

C 或 c：商标图案的属性颜色值(0～255)，默认为 10(=00001010b，高亮度绿色)。

D 或 d：显示商标模块或.BMP 的图形文件(测试之用)。

I 或 i：指定输入的模块文件 / 图形文件名。

L 或 l：指定字符的高度(8、14 或 16)、默认值为 14。

O 或 o：输出模块文件名。默认值为输入文件名，同时扩展名为.BIN。

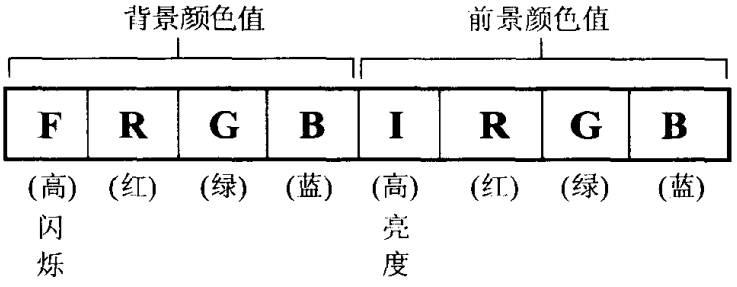
V 或 v：预视 OEM Logo(OEM 商标图文模块) 的信息。

X 或 x：从左上角起算，商标图文模块的水平显示位置(X 值从 0～79)。

Y 或 y：从左上角起算，商标图文模块的垂直显示位置(Y 值从 0～24)，默认值自动计算决定

以下是所作出的LOGO图形模块文件的资料格式：

偏移值 offset	字段变量	注　　释
00	db"GL"	供AMI BIOS识别的注记(Graphic Logo?!)
02	db x	图形左上角显示的起始行数(0<x<79)
03	db y	图形左上角显示的起始列数(0<y<24)
04	db wx	显示的行数(相当于.bmp图片的宽度除以八取整数值)
05	db ry	显示的列数(相当于.bmp图片高度除以8，14或16)
06	dbbg_fgColor	图形颜色值(BRGBIRGB)，默认值10(=00001010b，高亮度绿色)
07	db n	保留未用
08	.....	BMP转点阵字形数据起始位置



OEM Logo文字的属性

要注意的是整个转换有下列三大限制。

① 图形水平点宽度固定为八的倍数，垂直点固定为8，14或16的倍数。

② 必须是未压缩的单色(仅黑、白两色)的.BMP图形文件格式。

③ 只能以最多96个字母拼凑出来，每个字的宽度是八个点，字的高度则是14，15或16个点。

以 8×16 字为例，若宽度大约 128 个点，每八个点是一个字的宽度，因此每列可以显示 128/8=16，也就是 16 行字，此时只能显示 96 (个字) / 16(一列 16 行字) =6 列文字，而六列文字的高度是 6×16=96，此时这个图案的尺寸就是 28×96。

若采用 8×8 的文字格式，高度限制成 8×6=48 个点，这个.BMP 图片的大小就是 128×48 (或小于这个范围)。

若以 8×14 文字格式转换，则高度限制成 14×6=84 个点，这个.BMP 图片的大小就是 128(个水平点)×84 (个垂直点)，或低于这个范围。

因此厂商在设计自己的 OEM Logo 商标文件时，要熟记这样的限制。而在第 10 章硬件移植实战 DIY 中，我们会为读者示范一次更改商标图文件的做法。

ASL.EXE

ASL.EXE 是微软针对 ACPI 控制语言所开发的 ASL、AML 语言编译器。其语法如图 9-38 所示。

```
C:\627.10\SIS530\OEM\DEMO>asl /?
ACPI Source Language Assembler Version 1.0.11 [Jan 18 1999, 17:50:22]
Copyright (c) 1996,1999 Microsoft Corporation
Compliant with ACPI 1.0 specification

Usage:
asl /?
asl [/nologo] [/Fo=<AMLFile>] [/Fa=<ASMFile>] [/Fl=<LSTFile>] [/Fn=<NSDFile>] <A
SLFile>

      /?              - Print this help message.
      Fo=<AMLFile>    - Override the AML file name in the DefinitionBlock.
      Fa=<ASMFile>    - Generate .ASM file with the name <ASMFile>.
      Fl=<LSTFile>    - Generate .LST file with the name <LSTFile>.
      Fn=<NSDFile>    - Generate NameSpace Dump file with the name <NSDFile>.
      nologo          - Supress logo banner.

C:\627.10\SIS530\OEM\DEMO>
```

图 9-38　在 DOS 环境下执行 ASL.EXE

?	显示帮助信息
nologo	Supress logo banner. 忽略程序商标的显示
Fo=<AMLFile>	Override the AML file name in the DefinitionBlock. 定义的区块数据覆盖到AML语言控制码文件
Fa=<ASMFile>	Generate .ASM file with the name <ASMFile>. 产生一个文件名为ASMFile的.ASM型态的汇编语言源文件
Fl=<LSTFile>	Generate .LST file with the name <LSTFile>. 产生一个名为LSTFile的.LST型态的编译过程报表打印文件
Fn=<NSDFile>	Generate NameSpace Dump file with the name <NSDFile>. 以输入的NSDfile产生一个命令变量打印文件报告文件

ACPI接口协议与相关规范还在修订中，一般是笔记本电脑比较能完整呈现其技术内涵与实作的舞台，是笔记本电脑研发人员绝对需要搞清楚、弄明白的地方；一般台式计算机(主板)对于ACPI规范的实作项目并不多，仅应用到其中的ATX软件关闭电源、暂停到内存(Suspend to RAM)以及暂停到硬盘(Suspend to DISK)等项目。

由于前面也说过本书以台式计算机主板的BIOS Porting实作为主，对于ACPI的技术，仅在第1章与其他介绍源代码浏览的相关章节中有提到之外，关于ACPI的深入技术部分以及相关开发工具，可能要向AMI公司寻求协助，并通过微软网站(www.microsoft.com)或下列网址：

www.teleport.com/

FONT.EXE

FONT.EXE是多国语言字形点阵数据获取/转换程序，它会依照系统开发者的需求，到指定目录获取繁体中文、简体中文、日文、韩文等字形数据，然后生成一个包含字形点阵数据的FONT.BMP，然后再通过AMIBCP.EXE加入BIOS模块中。

目前 AMI BIOS 的中文、日文等多国语言的支持结构，以及整个外挂的模块设计与编译流程，甚至从 BIOS 内部模块，到工具程序都会有一些重大改变。目前多国语言的支持部分，AMI 公司并没有开放给 OEM 系统厂商 / 主板厂商来自行追加，而是以专案处理的方式进行，对此有中文或其他语言显示需要的厂商 BIOS 研发人员，请联系 AMI 公司来寻求技术上的协助。

RETRIV.EXE

RETRIV.EXE 是纯 DOS 命令模式下，针对 BIOS 影像文件做特定模块解压 / 切离工具程序。它的语法如下：

```
RETRIV ROM_FILE_NAME MODULE_ID MODULE_FILE_NAME
```

例如我们要把 1SIS5300.ROM 中的 Virus 防毒模块(ID=36h)分离成一个 TCAV.BIN 的文件，可以输入：

```
RETRIV 1SIS5300.ROM 54 TCAV.BIN
```

把 1SIS5300.ROM 中的 ID54(=36h)号模块，也就是 Virus 防毒模块，把它解压出来存为一个叫 TCAV.BIN 的二进制文件。

ROMSETUP.EXE

OEM 计算机厂商(或主板)研发人员，在写好、Porting 好一个 BIOS 之后，当然是很兴奋地想要赶快写到 FlashEPROM 芯片里头，然后开机试试看，不过在许多需要测试设计画面、选项的情形下，反复地修改、Porting、Flashing 一组 BIOS，然后重新开机进 Windows，是一件很浪费时间又累人的事情。

如果说有什么程序，可以在DOS(或Windows环境下的DOS模式)，去解压并模拟执行已经写好的BIOS程序代码，这样就可以在DOS的环境下联机操作，进行即时修改，那对于开发BIOS的工程人员来说是莫大的恩惠。而ROMSETUP.EXE就是一个这样的工具程序。

在使用之前，先将playrom.exe以及retriv.exe这两个文件拷贝到当前BIOS的计划目录(例如本书的C:\627.10\sis530\oem\lead\)，也就是要模拟执行的BIOS监控程序所在目录的位置，接下来输入romsetup *biosfile.rom* 即可。

以我们写好、Porting好的1SIS5300.ROM为例，它的界面格式是HiFlex v1.22，当我们拷贝playrom.exe、retriv.exe到相关目录后，直接输入ROMSETUP 1SIS5300.ROM，便会出现如图9-39所示的执行画面。

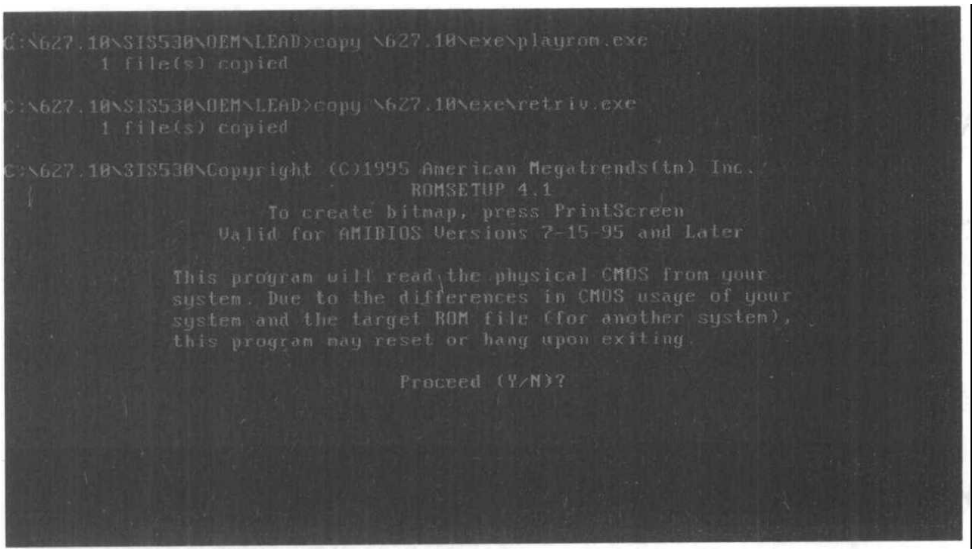


图9-39 执行后的画面

大意是说这个程序从你主板的CMOS内存读取参数值，如果你在模拟执行的这套BIOS监控程序，并不是针对你目前主板所设计(或者CPU、芯片组结构相近)的话，由于不同结构的BIOS、CMOS参数地址不见得都相同，模拟执行可能会有死机、停在某几个画面选项，或者结束仿真返回DOS时死机的情形。

按下 [redacted] 后，你可以发现真的可以在 DOS 环境(或 Windows 窗口的 MS-DOS 模式)下模拟执行 BIOS 的设置程序，显示出每个选项的画面(如图9-40、图 9-41 所示画面)。

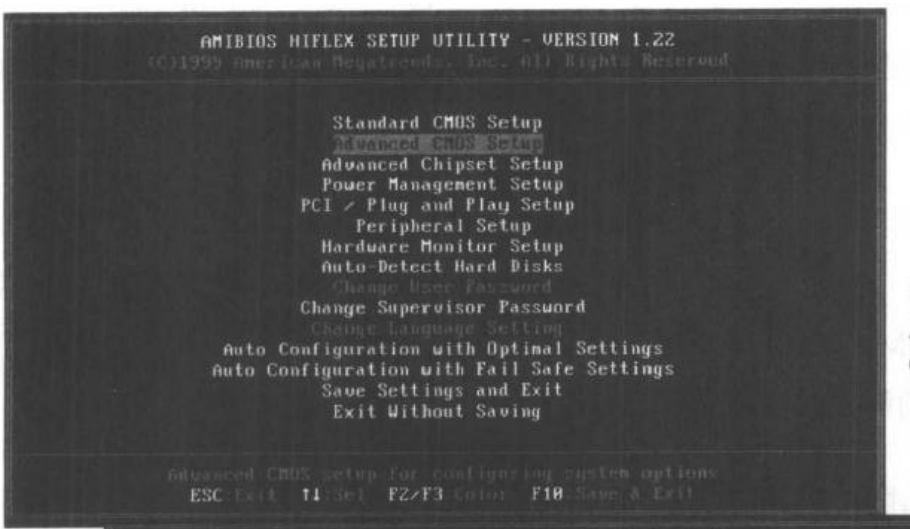


图 9-40 选择 Advanced CMOS Setup

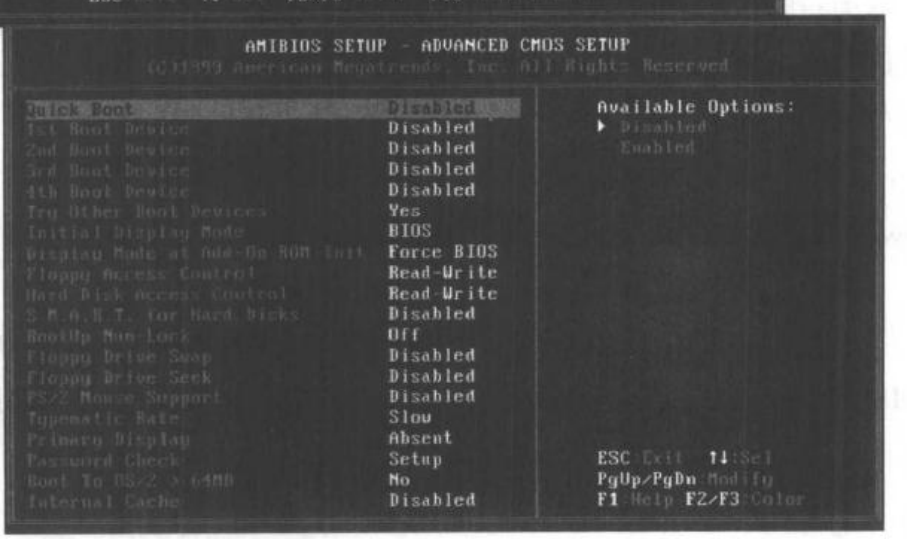


图9-41 进入后的画面

ROMSETUP.EXE 还有一项功能，就是按下 [redacted] (v4.1 版) 或 [redacted]+[redacted](II v1.0)两个热键按钮组合时，会将当前的 BIOS 画面，以 bmp 图形文件的格式，存成一个 a.bmp 文件(或其他指定的 xxxx.bmp)。这对于要编写 BIOS 手册的人员来说，可以获取 BIOS 设置时的画面，对于编写、排版上会有所帮助。

不过笔者检视所得到的*.bmp 图文件，却发现有时候存取的.bmp 图形文件格式不正确，无法被其他图像软件正确地识别出来。经笔者向 AMI 技术人员再三询问结果，由于 ROMSETUP.EXE 需要从被读入模拟执行的 BIOS 影像文件中，解压其中部分模块并调用执行，而不含 ML (MultiLanguage) 多语言支持模块的纯英文 AMI BIOS 影像文件，要做.bmp 图文件获取 / 转存时，可能会有不可预期的结果。

所以笔者的建议是，读者先在 Windows 环境下执行 MS-DOS 模式，然后切换到 DOS 窗口模式，把 retriv.exe、playrom.exe 拷贝到指定的目录，然后执行 ROMSETUP filename.rom，借助 Windows 的剪贴板，或其他图像软件的抓图功能来抓图就行了，本书不也是这样完成的吗?

最后，由于 ROMSETUP.EXE 还得搭配 Retriv.exe 与 playrom.exe 这两个可执行文件，为了操作上的方便，笔者简单地写了一个叫做 Showbios.bat，只要输入：

```
showbios filename.rom [ml]   ← 后面接 ml 表示这 BIOS 有 MultiLanguage 模块支持
```

这个批处理文件就会自动拷贝必要的文件到当前的计划目录，然后调用 ROMSETUP.EXE 来模拟 / 执行 BIOS 设定画面，跳出来之后会删除拷贝进来的执行文件。以我们写好、Porting 好的 1SIS5300.ROM 为例，就可以下这条命令：

```
showbios 1sis5300.rom
```

这样，要测试 BIOS 的设置画面、选项，甚至要抓图，都方便多了。

CHAPTER

10

硬件移植实战 DIY

经过了前面好几个章节的说明，相信各位对于 AMI BIOS 的整个源代码，从目录规划、执行流程、移植的关键模块 / 源代码部分等都应该有了充分的了解，为了让读者能进一步地体验这种深入 BIOS 移植与开发的乐趣，我们特地准备了由 AMI 公司所授权的 BIOS 源代码光盘，这份 BIOS 源代码主要是针对硅统科技 SiS530 芯片组结构所研发的版本，可以移植到各种以 SiS530 整合芯片组(Socket7 结构，插 Intel Pentium MMX、AMD K6-2 、K6- III以及 Cyrix 6x86、6x86MX 等处理器)设计的主板。

您想亲自体验那种编译好程序代码烧进 FlashROM 后的感觉吗？您想自己修改 BIOS 开机时那一成不变的画面吗？在看完下列警告之后，如果您对自己充满信心，也愿意承担一切的风险，那还等什么，跟着我们做就对啦！

以下的步骤，对于硬件不熟悉的生手而言，有一定程度的危险性，而风险的范围，可能从无法开机需要送修，到主板零组件因为插拔短路而造成的意外损坏，虽然笔者在文章中会详细叙述一切安全的细节，但难免不会有这种情况发生。如果您并没有硬件组装、自行升级、烧写 BIOS 的相关经验，建议您先熟读前面几个章节，将本章节视为观赏编译、烧写的实作绝学；如果您一定想做，也请与有硬件组装经验、做过烧写升级 BIOS 的朋友一起做，而对于一切可能发生的风险，我们已经尽了事先告知的义务，但恕不保证烧写本书所附源代码后，成功的绝对可行性与可能性，一切风险请您自负！

10-1 简介本书所示范的主板与其他配件

主板	最少一块 SiS530 整合型芯片组的主板(移植实作 / 兼程序编写平台)。 另备一块任意结构的主板，搭配 ROM 仿真器作为程序编写平台(非
处理器	Socket7 结构的处理器，不限品牌与速率(速度越快越好，建议用 K6- III+)
内存	建议最少 64MB SDRAM，128MB 以上的 SDRAM 是比较适合的。
硬盘	转速5400rpm以上的硬盘，建议最少4GB以上的容量，以备安装 Windows 2000以及 Windows 95，Windows 98，Windows 98SE 或Windows ME 等窗口操作系统。
操作系统	Windows 95以上的操作系统，建议 Windows 98SE(第二版)或 Windows ME ， Windows 2000 。
程序开发套件	MicrosoftMacroAssembler6.11(MASM6.11)，或其他含有此开发套件的程序开发工具。
辅助硬件套件	1 .能显示 POST Code 的除错适配卡，ISA 或 PCI 接口(必备)。 2 .最少一块 2Mb(256KB)的 28/29 F002 系列的 FlashROM 芯片，多多益善。 3 .ROM/SRAM Emulator(ROM 仿真器，如照片)非绝对必要。能拥有者最好。 4 . ROM 芯片辅助插座(如照片,非必要)，为无ROM仿真器与第二块主板时的替代方案。

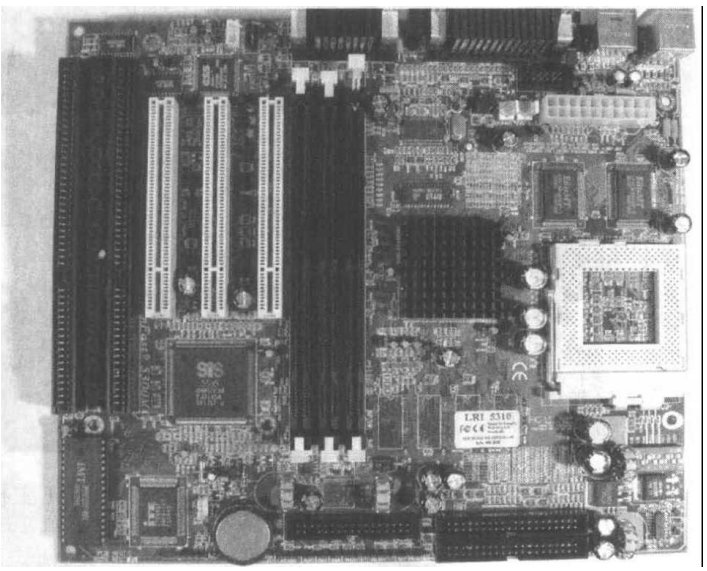


图 10-1 本书所示的丽台(Leadtek) Winfast5310MA 主板

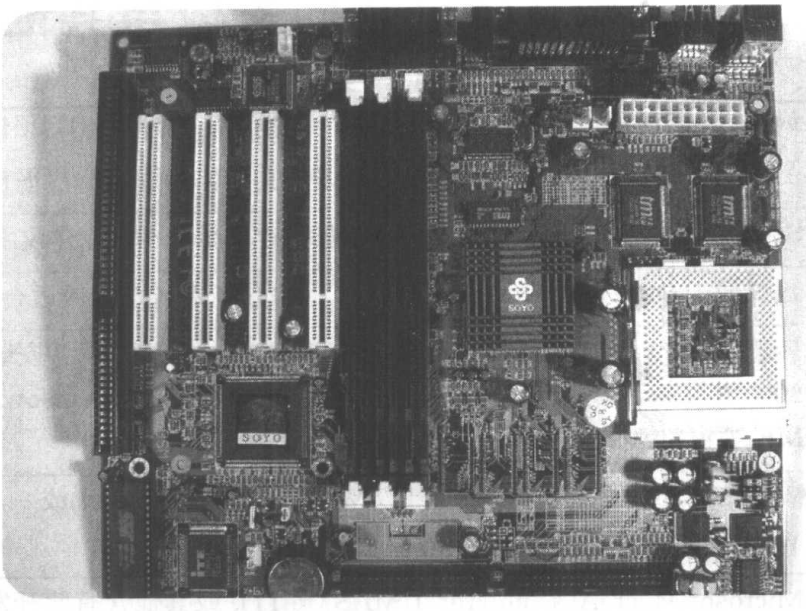


图 10-2　作者所成功移植的梅捷(Soyo) SY-5SSM 主板

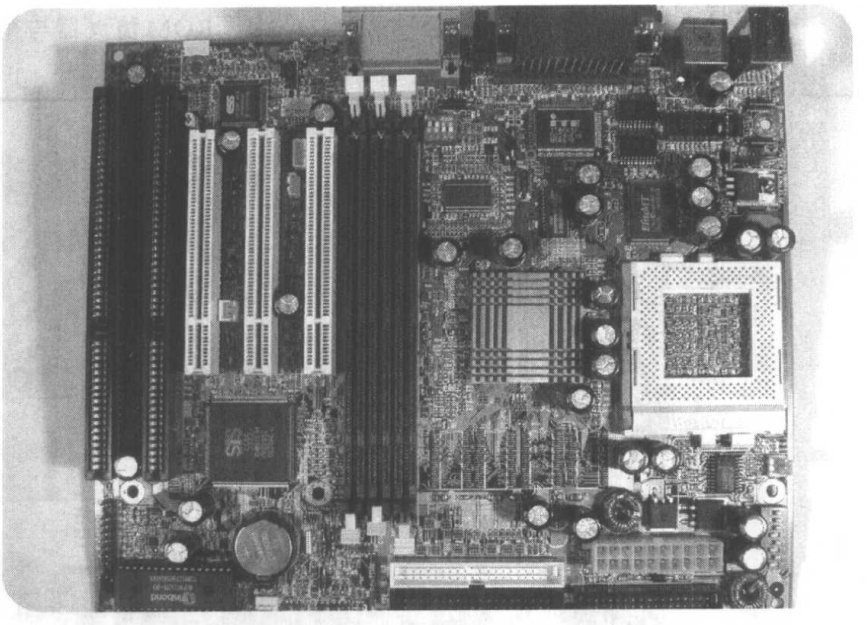


图 10-3　作者所成功移植的梅捷(Soyo) SY-5SSM 主板

读者可能会觉得很奇怪，怎么这些主板都是用 AWARD BIOS 的内存芯片呢？没错！不可讳言的是 Award BIOS 在台式计算机市场上的占有率还是比较高一点，不过随着本书出来之后那就不一定了。

事实上我们在真正实作时，原厂的 AWARD BIOS 芯片绝对要保留备用着，读者千万不要为了怕麻烦，而直接把源代码编译完成之后保存，万一写错了、写坏了，你这块主板可就没有可正常开机的 BIOS 了，得要送到原厂处理啦！

最后，我们要再提醒您一遍，就是本书光盘所附的 AMI BIOS 源代码，是针对 SIS530 芯片组所开发的特定版本，只能编译并移植到同为 SIS530 芯片组所设计的主板上；绝对不可以直接编译、烧写到其他芯片组所设计的主板：如 Intel、威盛 VIA、扬智 ALi 芯片组的主板上，如果引起主板无法开机所导致的维修问题，一样请您风险自负，我们恕不负责！

那要是像同一家芯片组厂商(如硅统)所设计的芯片组结构的主板呢？例如同样为 Socket7 结构的 SIS540 芯片组 / 主板，甚至更高级的 SIS630，SIS730 主板又会怎么样？

SIS630，730 主板上的移植是确定行不通的(源代码在核心层就要做很大的改动)，但是跟 530 结构相近的 540 又是如何？老实说笔者没测试过，也许移植成功机率会比 630，730 要高一些，如果您功力高，也许可以凭着这份光盘的源代码，再在网络上下载一些 SIS540 的技术资料，也许还真的有办法可以自行作出 SIS540 的 BIOS 也说不定喔！

完整以及解决问题的开发平台

其实在 BIOS 公司(或主板 BIOS 的研发部门)内，他们大多是采用下列这种两部机器互连的方式，一是称为程序开发端(Programming Appliance)，BIOS 源代码以及编译程序、除错程序都放在程序开发端这里，BIOS 监控程序要烧写、测试的那部分(主板)称为移植端(Porting Appliance)。程序开发端的机器是通过一种称为 ROM Emulator 的装置，以 RS-232C 串行口或 Printer Port 并行端口连接，另一端有扩展排线直接连到移植端主板上的 FlashEPROM 芯片的插座上，如图 10-4 所示。

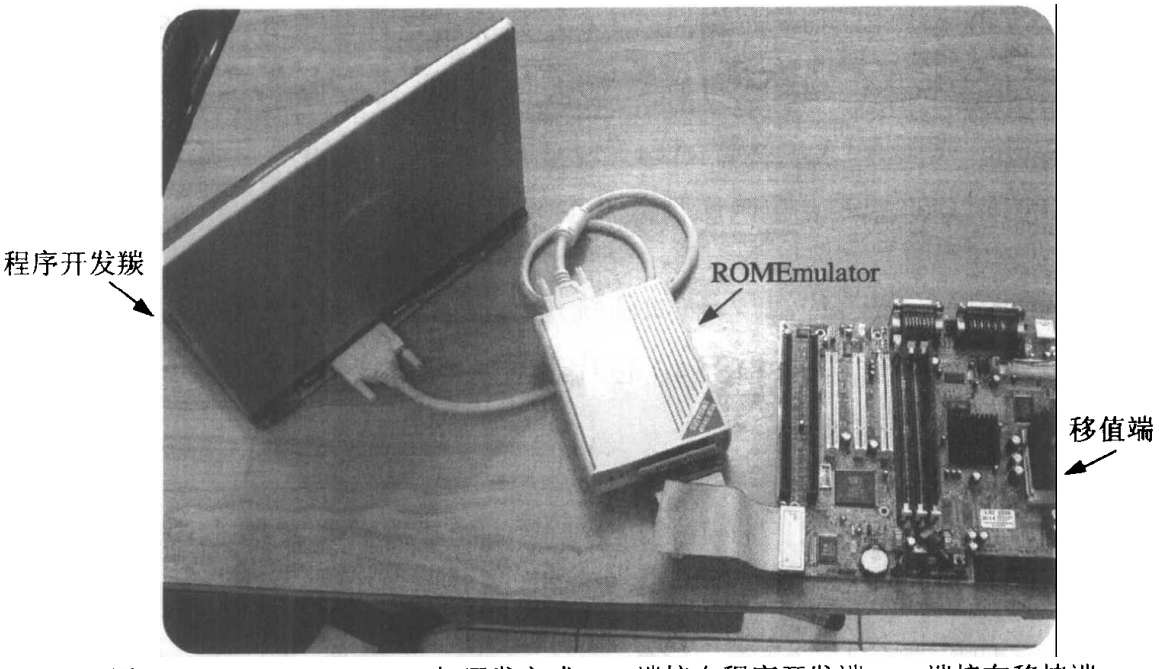


图 10-4　ROM Emulator 与开发方式：一端接在程序开发端，一端接在移植端

这样的好处就是在开发端写完、编译好程序代码之后，可以通过应用程序直接把 BIOS 监控程序送到 ROM Emulator，通过排线的连接，此时您就可以打开移植端的计算机，它所抓到的就是位于 ROM Emulator 的 BIOS 监控程序数据；万一开发不顺利，只要把移植端电源关掉，开发端的计算机还是可以直接修改程序而不会受影响的。

对于实际在开发、修改 BIOS 的工程人员来说，反复开机测试几十、几百次都是非常平常的事情，有这样的双重开发平台当然是最理想不过的，从开发端把 BIOS 编译好，马上送到移植端，开电源进行测试，整个开发流程直接、迅速而不拖泥带水；但是得多一部计算机的购置成本，而一组 ROM Emulator 系统也不便宜，因此对于只是想浅尝编写、修改 BIOS 快感的一般读者而言，有必要这么下重金去研究吗？

因此，就连笔者在内，大部分时间也是采取直接在移植端编写、修改与编译 BIOS 程序的单一平台作业。笔者是借助一种称为 DIP 芯片可插拔式的辅助插座，可以把 FlashROM 芯片放在上面，然后直接插在主板的 DIP 插座上，如图 10-5 所示。

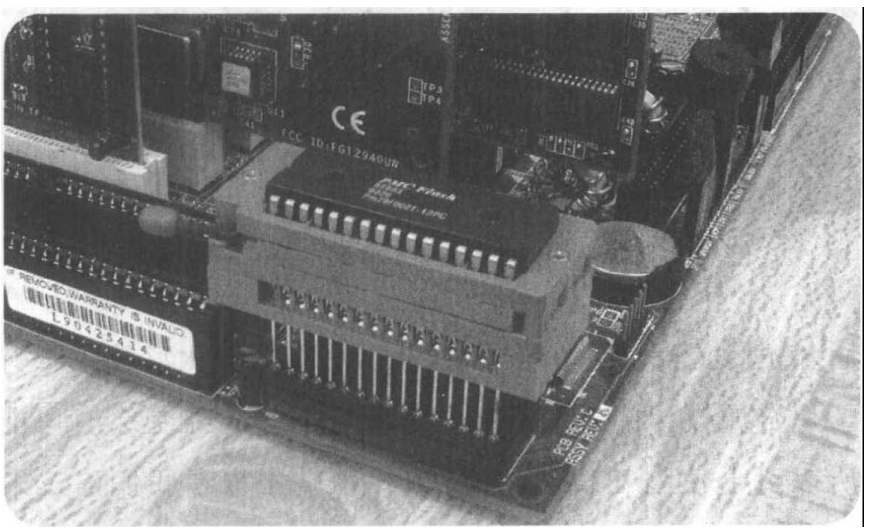


图 10-5　可插拔式 ROM 芯片辅助插座，适用于只有 DIP 规格，且单一主板开发平台

我们都知道主板有所谓的 Shadow RAM 的机制，会从 FlashEPROM 复制程序代码并且配置一块内存空间用来存储，而原先的 FlashEPROM 的内部数据就不再使用，所以我们在开机完成 (已经进入 DOS 或 Windows)后，只要把拉柄拉起来，就可以把原先插在上面的 Flash 芯片移走，换上别的要待烧写的 FlashEPROM 芯片。

另外，对于一些用 PLCC 封装的主板而言，我们可以用一种 IC 辅助起拔器，它的方式是顺着芯片两个角角的小沟槽，钩子从沟槽深入然后把 PLCC 封装的 FlashEPROM 芯片顺势拔出，如图 10-6 所示。

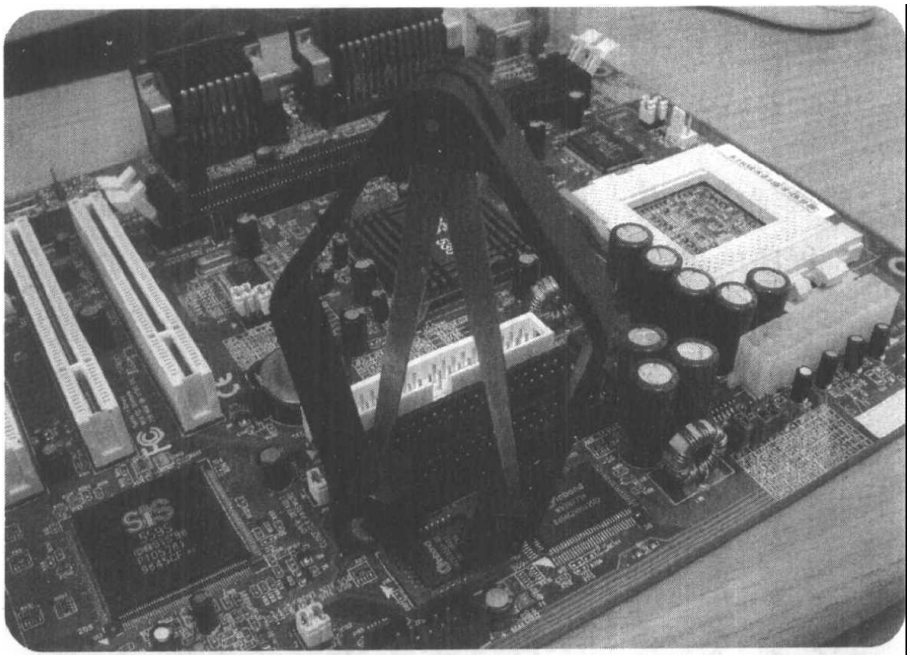


图 10-6　可拔取 DIP & PLCC 封装 Flash 芯片的 IC 起拔器

因为插入 PLCC 插座的 FlashEPROM 芯片不太好拔出，有些起拔器两旁的钩子是金属设计的，有时候一个拔取不慎，难免会有可能造成短路的风险，所以对于使用 PLCC 插座的移植端主板，我们建议采用上述的双平台开发模式。

当然笔者自己也试过，主板开机之后，照样用 IC 夹拔出 PLCC 封装的 FlashROM 芯片，但无论如何，如果读者一定要这么做，请买夹子两端钩子是陶瓷设计的(不导电，但价格较贵)。

还有对于生手来说，如果没有这种可插拔的辅助插座，笔者建议在未开电源下，先把 FlashEPROM 芯片拔起来，再原位对齐放回 DIP 或 PLCC 插槽，稍微用点力按上，以能够导通电路、计算机一开机还读得到 BIOS 芯片的内容为原则；此时松松接触的 BIOS 芯片，可以有助于您待会更换另一块 FlashEPROM 芯片，能够一次 OK。

为了安全起见，我们提醒您一定要留存好原厂可开机的 BIOS(不管是不是 AMI、AWARD 或其他厂商)芯片，不只是存盘而已，而是连整快 FlashEPROM 芯片带 ROM 一起保存以备用；万一开不了机时，这块唯一被验证过且能正常开机的 BIOS 芯片，就成为您这次实战之旅中保命的仙丹妙药。

10-2 建立必要的目录、相关文件与路径

这部分比较简单，基本上照着第 4 章的讲解，以及第 6 章第一、二节的说明，功力较高深的读者，应该可以练习自行尝试建立一些必要的目录、相关文件与路径，然后从附赠的光盘中，选择要复制的目录夹与文件。

再详细一点的安装程序，请照着下列四项步骤进行：

① 将整个 627.10 目录复制到工作硬盘

取出本书所附的光盘，将里面的 627.10 整个目录中的内容，一并复制到您要开发或移植的平台主硬盘 C:或 D:。此处建议采用 C:，因为本文范例以及部分批处理文件都是以 C:为工作磁盘驱动器。

这部分如果一切正常，就会出现如图 10-7 所示的整个目录结构。

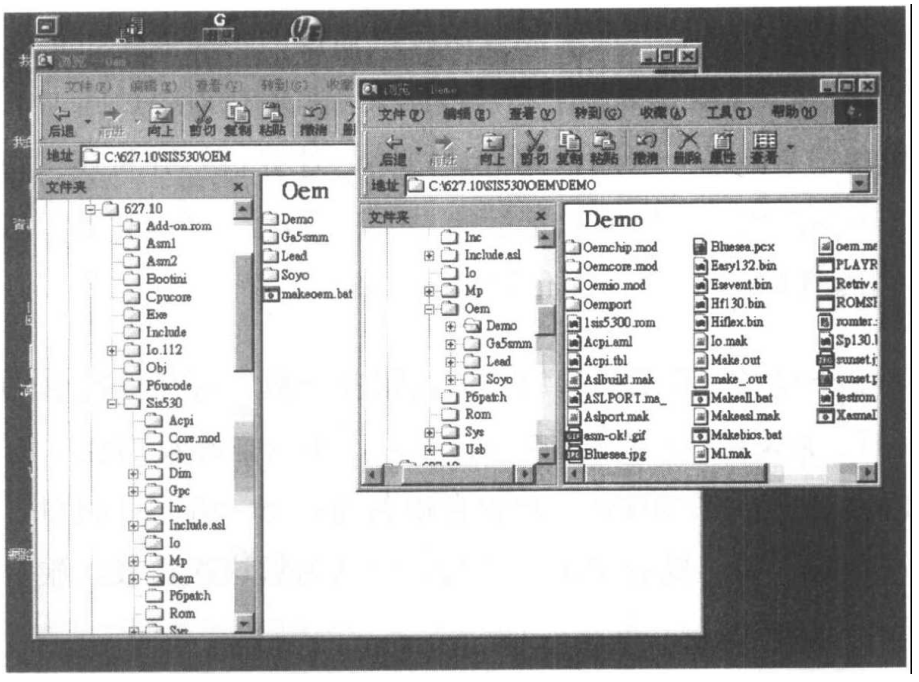


图 10-7　安装到硬盘之后的整个 AMI BIOS 目录结构

详细的目录设计与说明，读者可自行参考第 4 章第一节。

② 安装 MASM

AMI BIOS 大体上是用汇编语言(Assembly Language) 编写而成，所以自然要借助编译程序(Assembler)，这里我们建议使用微软的 Macro Assembler (MASM)，现在采用的版本是 6.11 版(目前最新的版本已发展到 6.14 版)，相关细节请看第 4 章第三节(※ 4-3)的说明。

总之要安装好 MASM 程序发展工具套件，才能进行下一步。

③ 安装ASL(ACPI Source Language Assembler)

如果您要移植、实作的主板，需要 ACPI 功能的话，那么就得借助 ASL.EXE 这个针对 ACPI ASL 以及 AML 语言编译的工具程序。取得方式一样请参考第 4 章第三节(※ 4-3)的说明。

④ 切换到 DEMO(或其他厂商 / 产品目录层)，实际编译看看

一切就绪之后，进入到 MS-DOS 模式，并切换到\627.10\SIS530\OEM\DEMO 或其他厂商层，输入 MAKEALL 或 MAKEBIOS 的指令，如果整个编译过程非常流畅，会出现如图 10-8 所示的画面。

图 10-8　指令执行后的画面

出现如上述有 BIOS 模块信息的画面，以及!! Module Manipulation Sucessful!! 的信息，那就表示目前您这套 AMI BIOS 目录结构设置还不错，目录、路径都对了，没有缺少文件。

若屏幕画面除了出现上述 BIOS 模块信息画面，最后几行出现了 filename not found 的信息，那应该是 AMI BIOS 在 PATH.MAK(位于 SIS530 芯片层)，将编译好的 BIOS 监控程序加载 ROM Emulator 的工具程序 LR.EXE 没有找到，如果读者手边没有这样的程序，对于这行找不到 LP 工具程序的错误信息可予以忽略。

若是读者有其他的 ROM Emulator 开发工具，或者要修改工具程序的名称，只要在 \627.10\SIS530\PATH.MAK 中，找出下列这段文字：

EMULATOR=LR

改成

EMULATOR=(你的ROM EMulator 的加载工具程序主文件名)

即可。

要是没有出现上述画面，而是在编译的过程中跳出来，那你可能要从目录、路径以及必要的编译工具程序一一检查，最后才是怀疑到程序代码是否有问题，若真有问题，就来信询问笔者吧！笔者的电子信箱为 chenbros@gcn.net.tw 。

一般说来，笔者是建议您在 OEM 目录下的 DEMO 目录去测试，而 DEMO1、DEMO2 以及 DEMO3 是笔者自己成功移植后所留的源代码，供读者们参考；当然您也可以用 MAKEOEM [vendor]或 MAKEOEM [product] 自己取个厂商 / 产品名称，然后在那个厂商 / 产品目录层去编译。

基本上这样就完成了，但是此时还不可以烧写喔！

为什么呢？因为还没针对所要移植的计算机主板的硬件特性，做细致的调校与修改呢！

10-3 开始进行 Porting 作业

此处我们先进到\627.10\SIS530\OEM\DEMO 示范目录层。除非有特别注明，要不然我们的实作目录，一切以 DEMO 这个目录为基准。

修改 OEM.MAK 以及 IO.MAK

找出移植端的主板，上面的 Super I/O 芯片、FlashEPROM BIOS 容量大小，然后分别做指定。这部分第 4 章第四节、第五节(※4-4、4-5)都有很详尽的说明。

以本书所示范的主板，查出它的 Super I/O 是联扬半导体的 ITE 8661F，所以我们打开当前 DEMO 目录下的 IO.MAK，找到下列几行：

```
#-------------------------------------------------------------------------------#
#        ONLY ONE IO SHOULD BE SELECTED                                         #
#-------------------------------------------------------------------------------#
AIP        =0        # 1 = Include Intel AIP                                    #
ALI5105    =0        # 1 = Include ALI 5105                                     #
ALI5113    =0        # 1 = Include ALI 5113                                     #
ALI5119    =0        # 1 = Include ALI 5119                                     #
ALI5123    =0        # 1 = Include ALI 5123                                     #
CT735      =0        # 1 = Include CT735                                        #
DSCH20     =0        # 1 = Include Dallas DS83CH20                              #
GM803B     =0        # 1 = Include LG Semicon GM82C803B                         #
GM803C     =0        # 1 = Include LG Semicon GM82C803C                         #

ITE8661    =1        # 1 = Include ITE 8661                                     #

ITE8680    =0        # 1 = Include ITE 8680                                     #
NAT303     =0        # 1 = Include National 303                                 #
NAT306     =0        # 1 = Include National 306                                 #
```

请仔细检查除了 ITE8661 字段有设为 1 之外，有没有其他的字段被误设为 1，有的话千万记得要改为 0，因为这一群 Super I/O 芯片字段中只能有一个字段设为 1。

至于 BOOT_KBC_RTC 以及 BOOT_FDC 两个字段都设成空白，不指定任何 SuperI/O 芯片，让默认值自行决定。

OEM.MAK 部分，请找到下列片段，检查并做下列更改：

```
BIOS_SIZE          =2                  ←— 指定 BIOS 容量为 256KB
SOFT_OFF           =1                  ←— 启用 ATX 软件电源开关功能
ACPI_SUPPORT              =1           ←— 启用 ACPI 功能支持
ONCHIP_LM78               =1
LM78_BASE_ADDRESS  =295h
LM75_SUPPORT       =1
LM75_SLAVE_ADDRESS =9eh
SILENT_BOOT               =0   ←—目前先不使用 Silent Boot 全屏幕图形静
                                  幕状态显示开机功能
GPNV_SUPPORT       =1          ←—平台结构支持 GPNV 功能
GPNV_FLASH                =1   ←— 将 GPNV 存储在 FlashEPROM 内部
EVENT_LOG                 =0
USB_SUPPORT               =1        ←— 指定启用 USB 功能并提供相关支持

SUI_FILE                  =hf130.bin    ←— 以 HiFlex 1.3 版作为 Setup 接口
#SUI_FILE                 =easy132.BIN
#SUI_FILE                 =HF122.BIN
#SUI_FILE                 =Win29.BIN

#SIGN_ON_MSG1     ='AMI BIOS for SiS 530',0  # sign on message
```

搜集PCI资料以及做PCI IRQ Routing

照道理，这一部分应该是BIOS研发人员跟主板线路设计人员需相互配合的项目，因为只有原主板线路设计者，才最了解PCI扩充槽、On Board PCI芯片/装置的中断信号顺序与线路设计。即使有些小厂对这种lay线路以及BIOS都是单独由一人负责，线路是您自己设计的，照理哪组PCI扩充槽接INTA#的线路顺序，您应该最清楚。

只不过在业界有个不传之密，就是很多时候为了争取快速开发主板上市的时间，系统平台厂商(处理器或芯片组厂商)会提供所谓的“工程参考样板(Engineering ReferenceBoard，也就是俗称的公板)”，如果是从公板参考、修改来的正式主板，只要结构跟公板相去不远，往往是拿公板的BIOS源代码，修改厂商商标以及一些设定选项，也就这样出货了，细心一点的，也是参照公板的PCI IRQ Routing先填上去，日后再做修改。总之，这些对于身处业界的开发工程师而言，技术资料以及参考资料的获得都不是很困难。

但是，对于稍有汇编语言基础，却还未有机会在业界大显身手的各位读者(我想这类读者应该不在少数)，除了本书以及光盘所附的AMI BIOS源代码，要从市面上弄来一块SiS530芯片组的主板就不太容易了，读者又怎么会晓得这块主板的Lay out设计呢？去跟厂商要线路设计图，您想厂商会给吗?(别傻了，商业机密呀!)

为了大多数处于业界之外的一般读者着想，笔者也特地向AMI公司挖宝，有一种不依靠原厂主板线路图而直接做PCI IRQ Routing的方法，这还是有IO Porting多年经验的AMI工程师的不传之密呢！

找出一切可能的 PCI 适配卡

改好 OEM.MAK 以及 IO.MAK 之后先关机，并关掉电源。

接下来要做的事情，就是尽量去找、去借一些 PCI 适配卡，您要移植的主板有几个 PCI 扩充槽，您就要插上几张 PCI 适配卡，这些适配卡只要有占用 IRQ 资源的就可以用，功能重叠也无妨(例如您插上两个网卡或两张 SCSI 卡) 。

一般 SiS530 主板会设计有三组、最多四组的 PCI 扩充槽，所以要准备三到四块 PCI 适配卡，如图 10-9、图 10-10 所示。

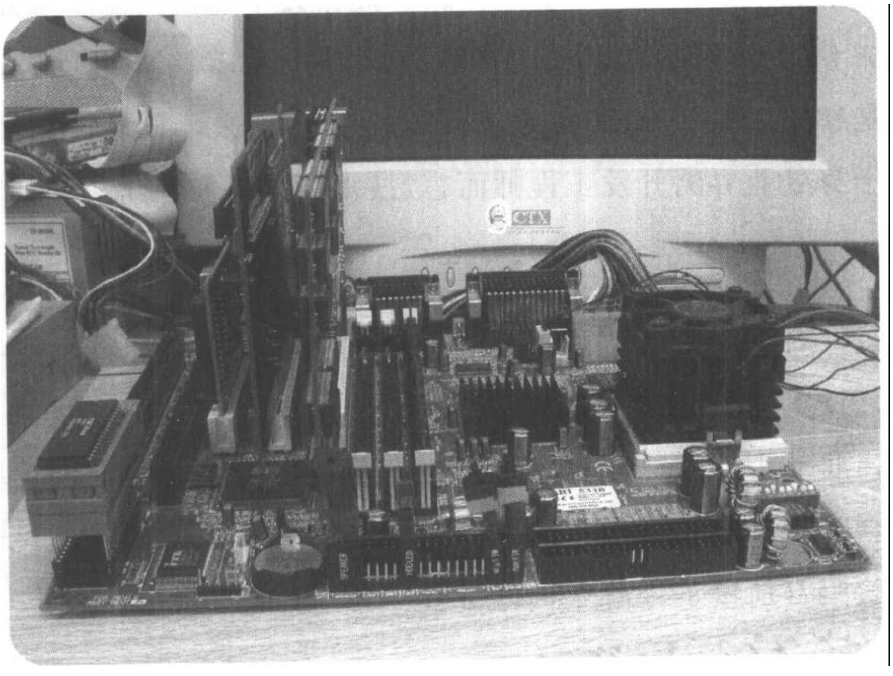


图 10-9　本书实作的丽台主板三组 PCI 插槽插满的情形

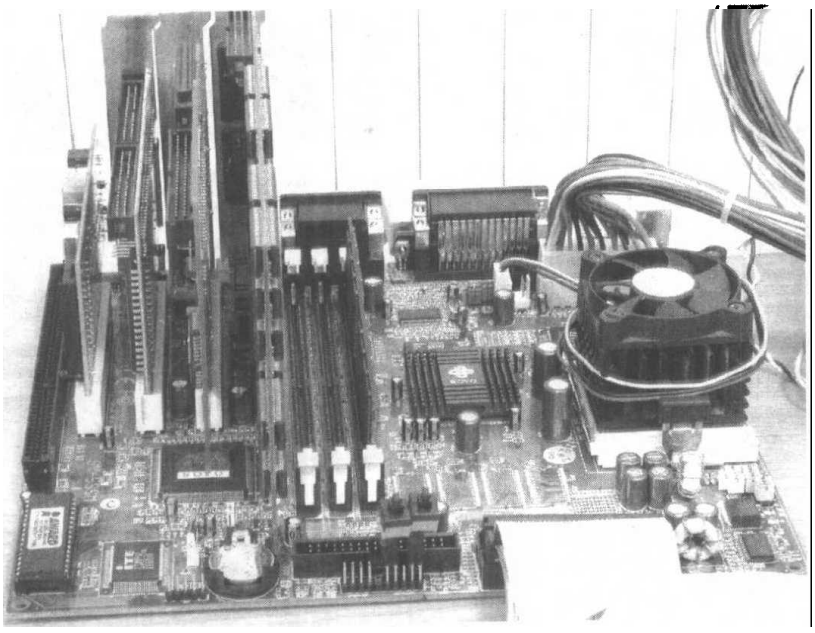


图 10-10　笔者另外在梅捷主板插满四组 PCI 插槽的情形

第一次开机时，由于我们还没搜集到主板的 PCI IRQ Routing 以及其他的硬件资料，AMI 版本的 SiS530 BIOS 又还没编译、调校好，在这里，我们还是先以原厂 AWARD BIOS 芯片来开机。

若您安装了 Windows 98,98SE，在 Startup Menu 出现时按下 F8 钮，然后选择 Command Prompt，进入 DOS 模式。

接下来请切换到\627.10\EXE 目录(若已经在 AUTOEXEC.BAT 设好工具程序的路径，亦可省掉这个步骤)，然后请执行 RU.EXE，此时 RU 会警告目前的系统并非使用 AMI BIOS，按 Esc 忽略掉这个警告画面后，按下 F6 键指定列出所有 PCI 装置的情形(如图 10-11 所示)。

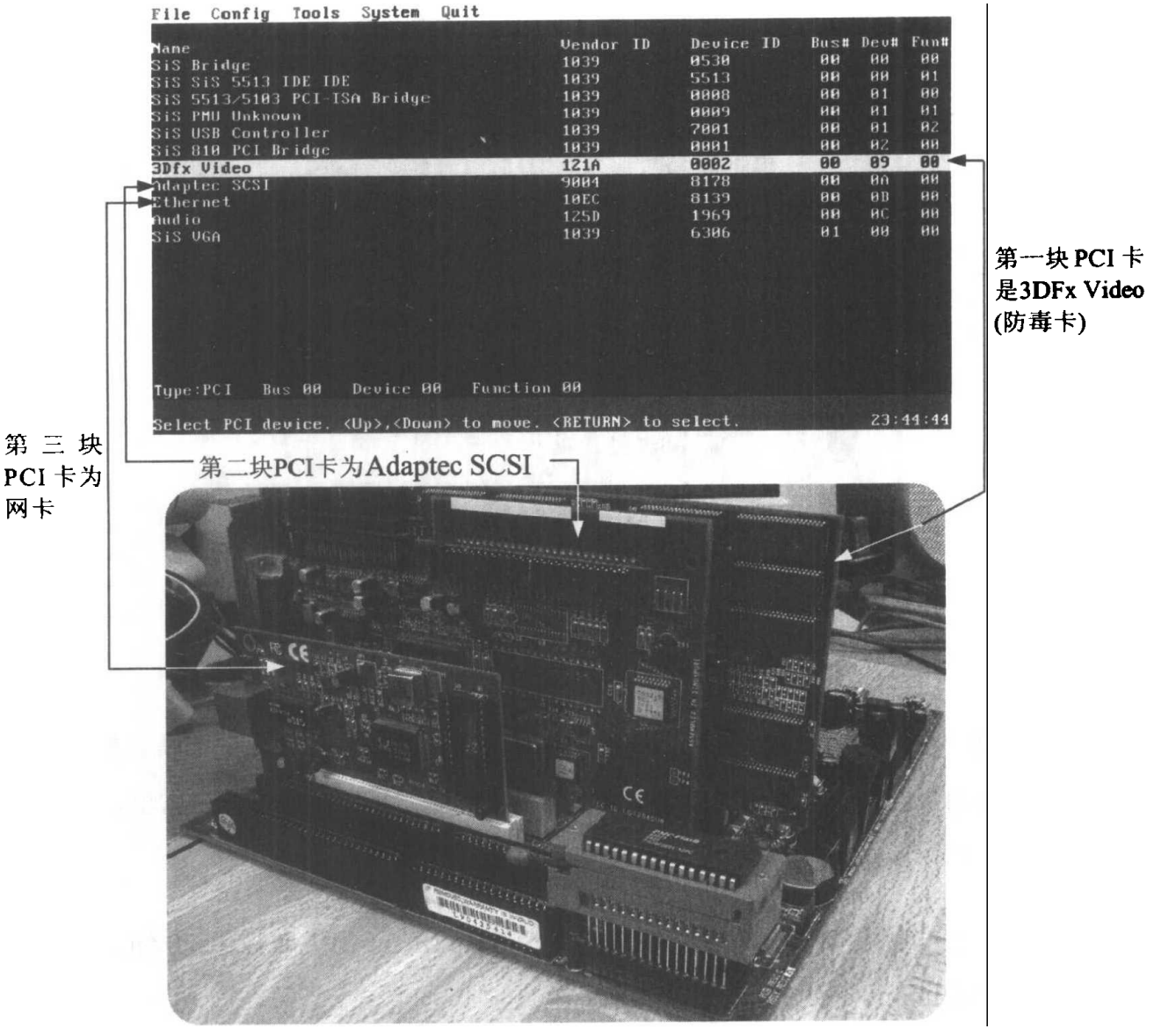


图10-11 通过 RU 列出所有 PCI 装置的组态数据 (此处是丽台 3PCI+2ISA+PCI SoundChip 组态)

把 RU 所列出的数据抄下来，或者利用抓图工具也是一种不错的方式。

此处我们发现要列入 OEMRPCI.ASM 的第一个项目是 SiS USB Controller (Bus 0, Dev 01)，第二个项目是 PCI 810 Bridge(北桥芯片本身，Bus 0, Dev 02)，第三个项目就是插在最右边第一组 PCI 的 3Dfx video(防毒卡，这里就是第一组 PCI 槽，占 Bus 0, Dev 09)，第四个项目就是插在三个 PCI 槽中间的 Adaptec SCSI(AHA-2940UW)适配卡，这代表第二组 PCI 槽，占用 Bus 0，Dev 0A(也就是 Device11)，最左边 PCI 槽所插的是网卡 Ethernet，占用 Bus 0, Dev 0B (Device 11)，最后一个项目为 Bus 0，Dev 0C(也就是 Device12)是 Audio，这代表 On Board Audio 芯片是最后一组 PCI Device。至于 SiS VGA 是 Bus 01, Dev 00，严格说起来它是内含在 PCI 810 Bridge(北桥芯片内)，此处我们先不加入这个项目为最后一项。

接下来先关机，把这些多余的、占据 PCI 扩充槽的适配卡全部移走，该还人的还人，该丢的丢(joke)，只留下必要的适配卡(如显卡、网卡等)，当然也要记得插上一块 POST/Debug Card，有助于待会儿编译、测试 BIOS 时用。

因为我们还没有将这些新搜集的组态信息，写到源代码并重新编译出一个能用的 AMIBIOS ROM 文件，所以目前还是先以原厂的 BIOS 芯片(如 AWARD BIOS)开机。接下来我们就将这个 RU 所侦测到的资料，对应成要写到 OEMPORT\OEMRPCI.ASM 的部分(★代表跟上一行连在一起)。

rth_pci_device_limit db 6 ；下列表格总共六个项目(有多少项则填入多少个)
rth_pci_device_count db 6 ；实际使用的装置项目

.....

rth_pci_irq_routing_table label byte ；PCI IRQ 绕线设定表格项目开始的标志

USB Device 1	Device Location	Int A Pin	Int B Pin	Int C Pin	Int D Pin	扩充槽号码	保留位
	PCI Bus DevNum Number (Shl 3)	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin		

pci_irq_entry<00h, 008h, 062h, 1101111010111000b, 00h, 0000000000000000b, 00h,
0000000000000000b, 000h, 0000000000000000b, 0, 0 >

AGP Bridge Device 2	Device Location	Int A Pin	Int B Pin	Int C Pin	Int D Pin	扩充槽号码	保留位
	PCI Bus DevNum Number (Shl 3)	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin		

pci_irq_entry<00h, 010h, 041h, 1101111010111000b, 42h, 1101111010111000b, 43h,
1101111010111000b, 044h, 1101111010111000b, 0, 0 >

Slot 1 AD#20 Device 9	Device Location	Int A Pin	Int B Pin	Int C Pin	Int D Pin	扩充槽号码	保留位
	PCI Bus DevNum Number (Shl 3)	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin		

pci_irq_entry<00h, 048h, 041h, 1101111010111000b, 42h, 1101111010111000b, 43h,
1101111010111000b, 044h, 1101111010111000b, 1, 0 >

Slot 2 AD#21 Device10	Device Location	Int A Pin	Int B Pin	Int C Pin	Int D Pin	扩充槽号码	保留位
	PCI Bus DevNum Number (Shl 3)	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin		

pci_irq_entry<00h, 050h, 044h, 1101111010111000b, 41h, 1101111010111000b, 42h,
1101111010111000b, 043h, 1101111010111000b, 2, 0 >

Slot 3 AD#22 Device11	Device Location	Int A Pin	Int B Pin	Int C Pin	Int D Pin	扩充槽号码	保留位
	PCI Bus DevNum Number (Shl 3)	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin		

pci_irq_entry<00h, 058h, 043h, 1101111010111000b, 44h, 1101111010111000b, 41h,
1101111010111000b, 042h, 1101111010111000b, 3, 0 >

Audio	Device Location	Int A Pin	Int B Pin	Int C Pin	Int D Pin	扩充槽号码	保留位
AD#23 Device12	PCI Bus DevNum Number (Shl 3)	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin	Chipset Bitmap of IRQs Reg Routableto Pin		

pci_irq_entry<00h, 060h, 044h, 1101111010111000b, 00h, 0000000000000000b, 00h,
0000000000000000b, 000h, 0000000000000000b, 0, 0 >

把上述这六个项目复制一份到下列标记：

ms_irq_route_table_cksm db 0 的下一行

(上述六个项目复制到这里)

.....
.....

然后把修改好的 OEMRPCI.ASM 保存到硬盘。

检查 ASLPORT.MAK（有 On Board 声卡或其他芯片的话）

若要移植的 SiS530 主板除了本身的北桥、南桥芯片、Super I/O 芯片之外，还有其他 PCI 接口的芯片，接下来我们还要进到当前的 DEMO 目录层下，把 ASLPORT.MAK 打开，并且找到下列的片段：

```
# Onboard PCI device 2
A_PCI_DEV2_IRQA      =
A_PCI_DEV2_IRQB      =
A_PCI_DEV2_IRQC      =
A_PCI_DEV2_IRQD      =0c.ffff,0  # Rounting for Sound chip
```

OnBoard PCI 音效芯片连到 INTD#

PCI 音效芯片为 Dev 0C (Device12)

如果还有其他 OnBoard PCI 芯片，则依此类推，然后回到 ASLPORT.MAK 这个文件。

其他源代码的修补

基本上这是对 AMI BIOS 编写、移植有深入经验的专业人员，想为自己手中计划的产品增加特色，如果您要在 BIOS 的设置画面做出选项或其他必要的修改时，可参考第 4、第 5、第 6 以及第 7 章的说明。

开始编译

现在一切都微调妥当后，切回 DEMO 这个目录为工作目录，在 MS-DOS 提示符号下键入：

C:\627.10\SIS530\OEM\DEMO>MAKEALL (或MAKEBIOS.BAT)

如果一切正常，会出现前面讲到的 BIOS 模块画面，以及!! Module Manipulation Sucessful !! 的信息。此时请以 AMIBCP 这个 BIOS 组态程序，打开刚刚编译、链接成功的 1SIS5300.ROM(如图 10-12 所示)。

C:\627.10\SIS530\OEM\DEMO>AMIBCP ISIS55300.ROM

仔细检查一下这块 BIOS 的预设参数值，最重要的是检查一下 IRQ Routing Table 正不正确，这涉及到这块 BIOS 有没有办法顺利通过 POST 并且顺利开机。

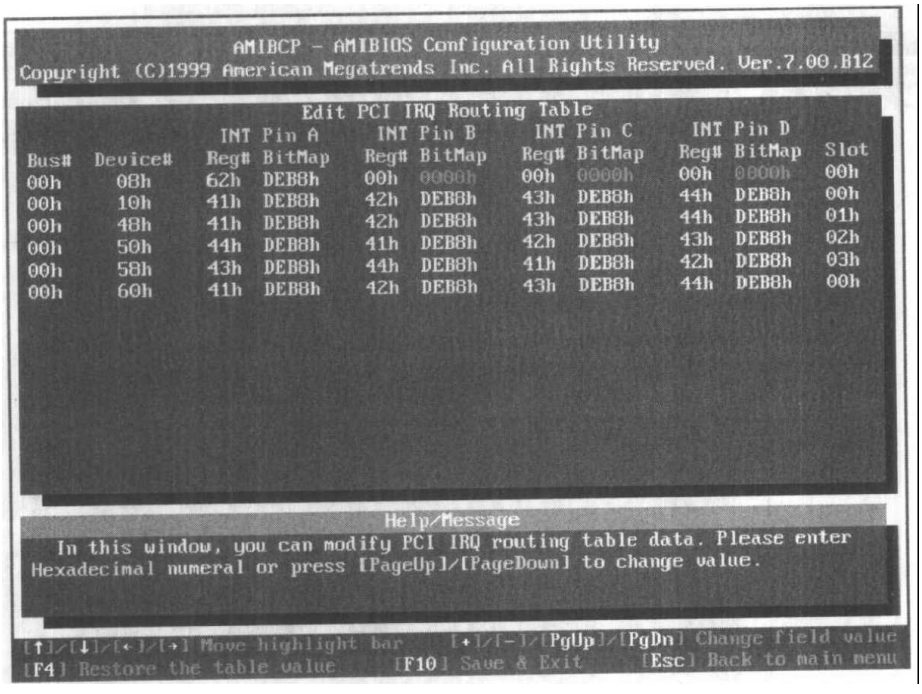


图 10-12　以 AMIBCP 进入 Edit PCI IRQ Routing Table 检查 IRQ 绕线设置

图 10-13 用 AMIBCP 检查 BIOS 组态，此时可写入您所要的预先设置值。

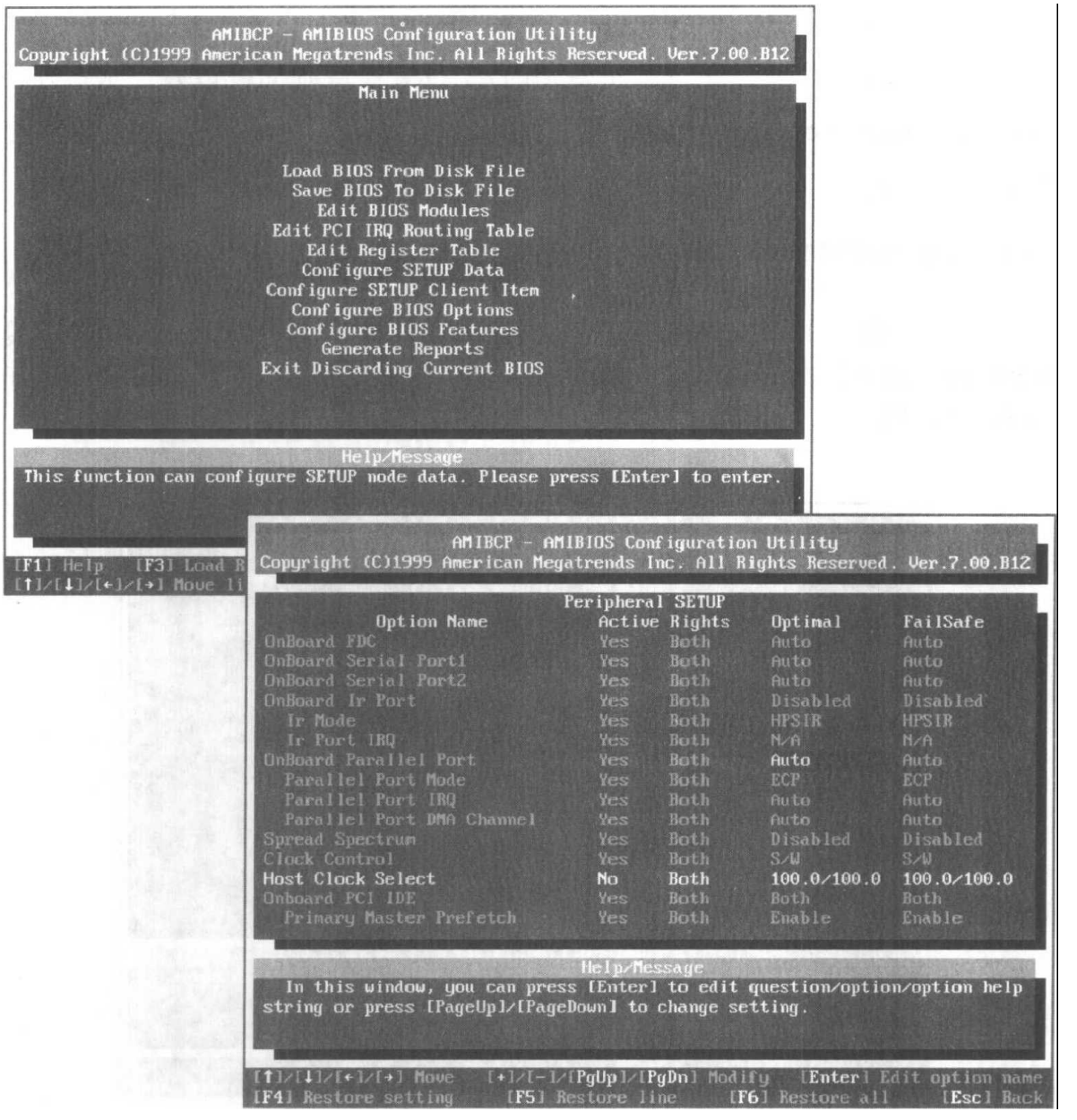


图 10-13　这部分可参考第 8 章第三节(※ 8-3)以及第 8 章第四节(※ 8-4)的说明

测试与实际写入

一切都 OK 之后，现在就可以准备要写入了。

提醒您在做烧写之前，先将 Windows 的屏幕保护程序 / 功能关闭，必要时连节电功能也先暂时关闭，而且不要把ROM文件放在以 USB 接口驱动的外接式硬盘、软盘或任何存储设备，这是因为 AMIFlash 程序在做 FlashEPROM 写入时，会关闭任何跟 USB 接口有关的驱动装置。

现在，如图10-14所示，把原厂 BIOS 芯片拔出，如果您不是像笔者用双平台配 ROM Emulator，或者有辅助拉柄的插座，前面不是提醒您原厂 BIOS 芯片不要插太深吗？现在您可以通过 IC 夹深入槽沟，一定要一次就拉起来，不要拖泥带水。

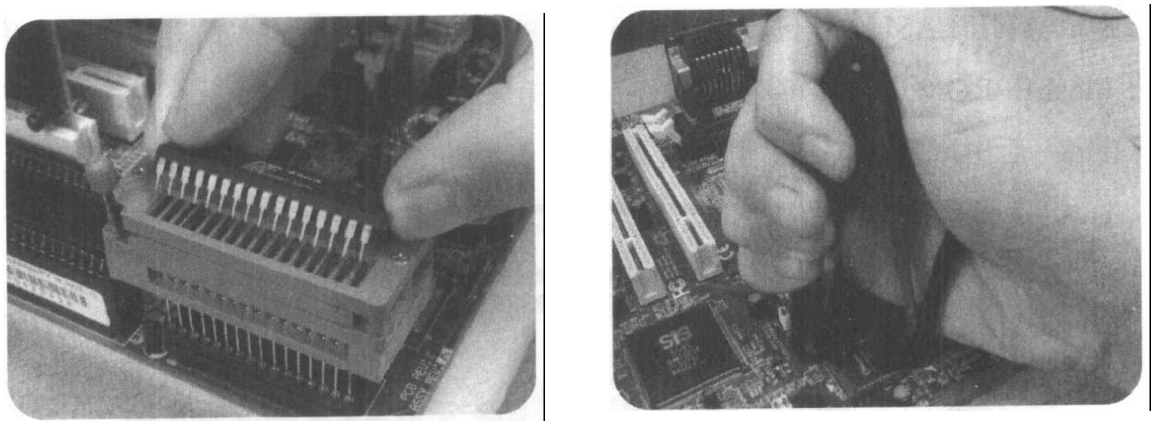


图10-14 把原厂 BIOS 芯片拔出，另外换上新的芯片

换上待烧入的 AMI BIOS 监控程序的空白芯片，接下来调入 Flash826 1SIS5300.ROM 。

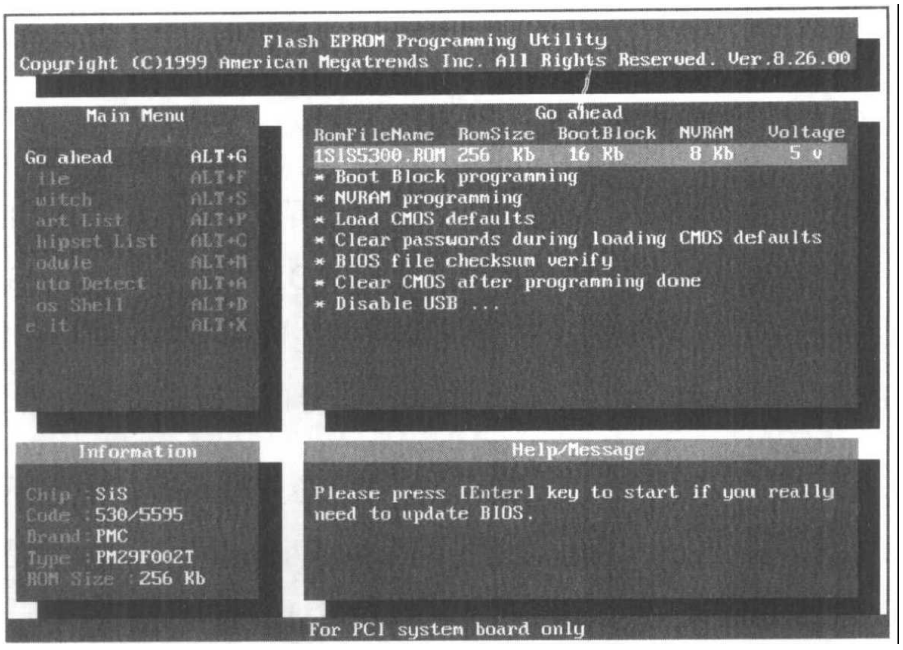


图 10-15　强制指定芯片型号并设置

如果侦测到的 FlashEPROM 芯片型号不对，先按下 [redacted] 键跳出，参考第 8 章 AMIFLASH 如何强制指定 FlashEPROM 芯片型号的方式，设定好正确的 FlashEPROM 之后，将控制光条列在第一项 Go ahead(进行烧录)选项，按下 [redacted] 后出现提示窗口，再按下 [redacted] 就开始进行烧写了，如图 10-16 所示。

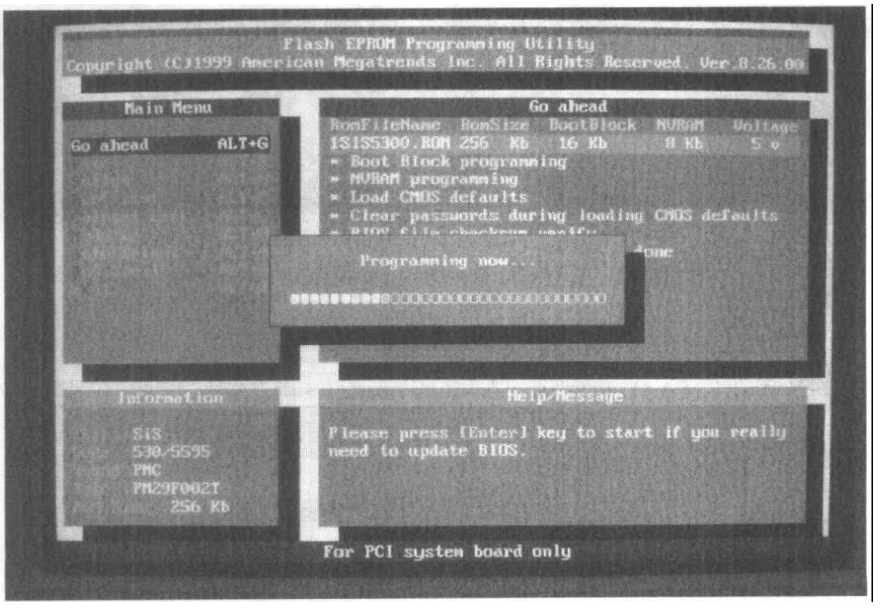


图 10-16　烧写的进程

在写入的过程中，千万不要移动鼠标，或者对其他 USB 装置做插拔的动作。写入完成之后，AMIFLASH 会出现如图 10-17 所示的写入完成信息，此时按任何键回到主菜单；或者等待几秒钟之后，AMIFLASH.EXE 会自动复位(Reset)开机。

AMIFLASH 烧写程序的详细操作，可参考第 8 章第二节。

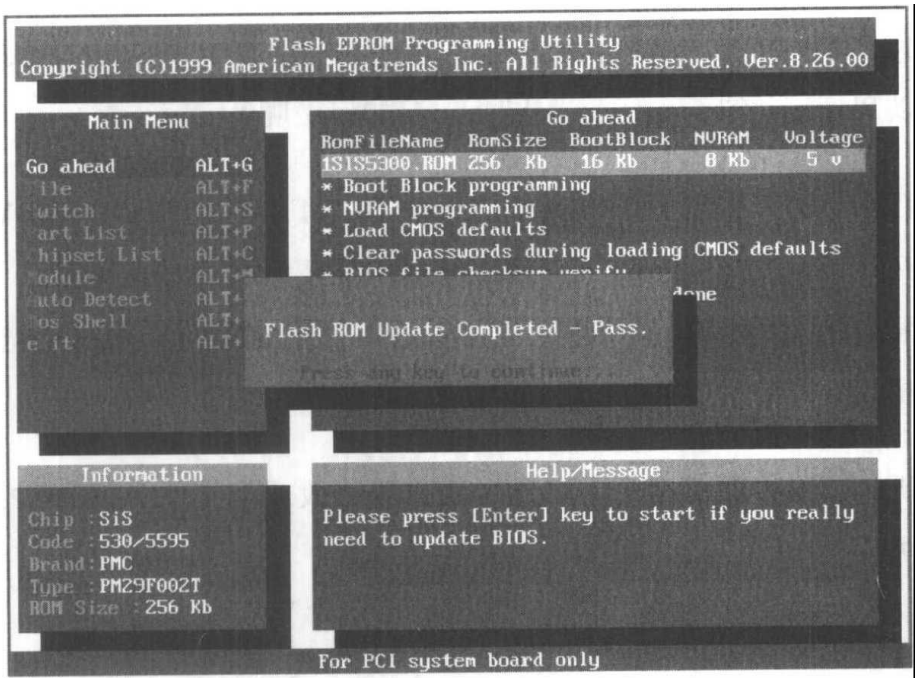


图 10-17　写入完成

进行到这里，希望您的编译、移植作业是成功的，如果不成功的话，应该会有错误的哔、哔声响，或者停留在哪个 POST Code 的画面，此时您就要参考附录的说明，回顾前面所介绍的章节，试着去除错、找出症结在哪里了。

10-4 修改开机的设置界面、商标图案与全屏幕画面

直接修改Setup 设置界面

您是否已经厌倦了像 Award BIOS 这种(AMI 则称为 Simple Setup)设置界面，想换点不一样的设置画面吗？假设我想从 HiFlex 或 Simple Setup 界面，换成类似 Phoenix 的 Easy Setup 界面，该如何做呢?

我们附带的有 SP130.BIN(Simple Setup V1.30)，HF132.BIN(HiFlex V1.32)，还有 EASY132.BIN(Easy Setup)，至于 Windows 设置界面文件(Winxx.bin)，由于年代久远，跟某些 BIOS 版本的连接还有未竟完美之处，在本书所附光盘中我们并没有提供。

改新的设置界面很简单，可通过源代码修改或直接以 AMIBCP.EXE 修改两种方式：

① 通过源代码修改

这个非常容易，只要在 OEM.MAK 找出 SUI_FILE=filename.bin，将这个字段文件名指定成自己要的设置界面文件，然后重新 MAKEALL 编译就可以了。例如把 SUI_FILE=hf130.bin 改成 easy132.bin，编译后烧到 FlashEPROM 芯片内，下次开机时的界面就改成 Easy Setup 的界面了。

② 通过AMIBCP.EXE修改

其实只要借助这些设置界面文件，就算有其他版本的芯片组，也是 AMI 的 BIOS ROM 文件，照样可以“改头换面”，换一个新的设置界面的脸孔。要怎么做呢？

以下假设我们所取到的 AMI BIOS ROM 文件，文件名为 AMIBIOS.ROM，首先我们以 AMIBCP.EXE 这个超强组态工具，打开这个 AMIBIOS ROM 文件：

```
C:\627.10\SIS530\OEM\DEMO\>AMIBCP AMIBIOS.ROM
```

选择 Edit BIOS Modules(编辑 BIOS 模块)，进入如图 10-18 的画面：

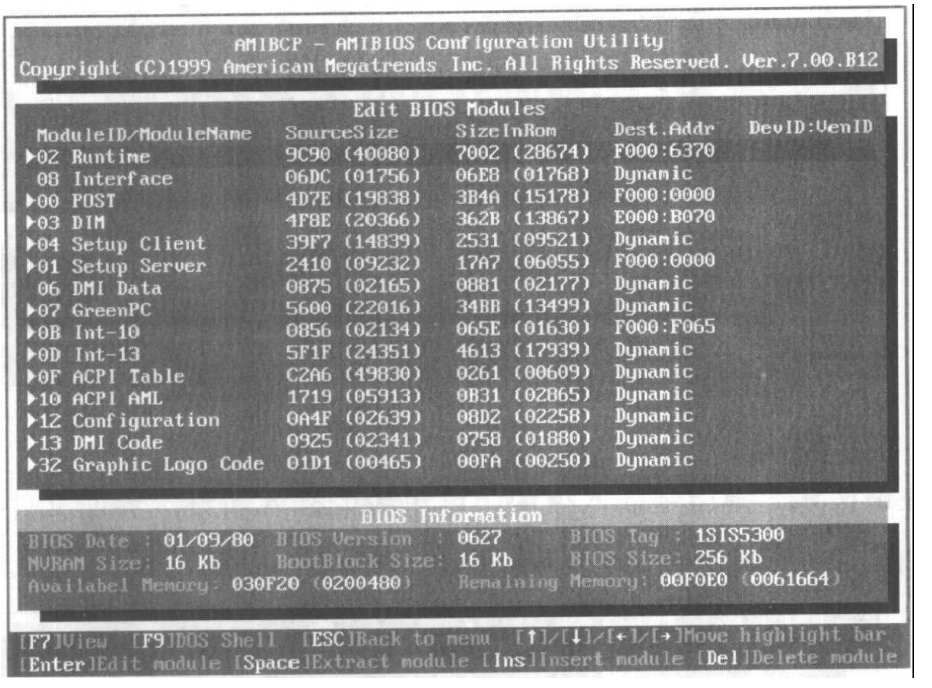


图 10-18　列出 AMIBIOS.ROM Rom 文件内部的所有模块

将控制光条移到 Setup Client(设定界面)，然后按 Delete 键准备删除 (如图 10-19)。

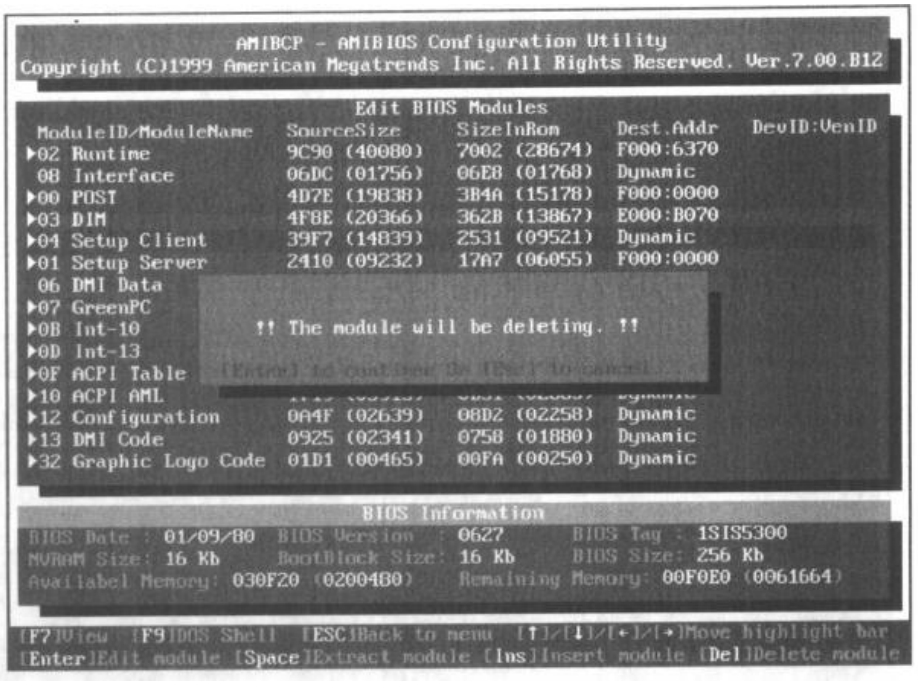


图 10-19　删除 Setup Client 这个模块

现在请按[redacted]键指定插入模块，这里我们插入新设定界面文件(easy132.bin)，如图 10-20 所示。

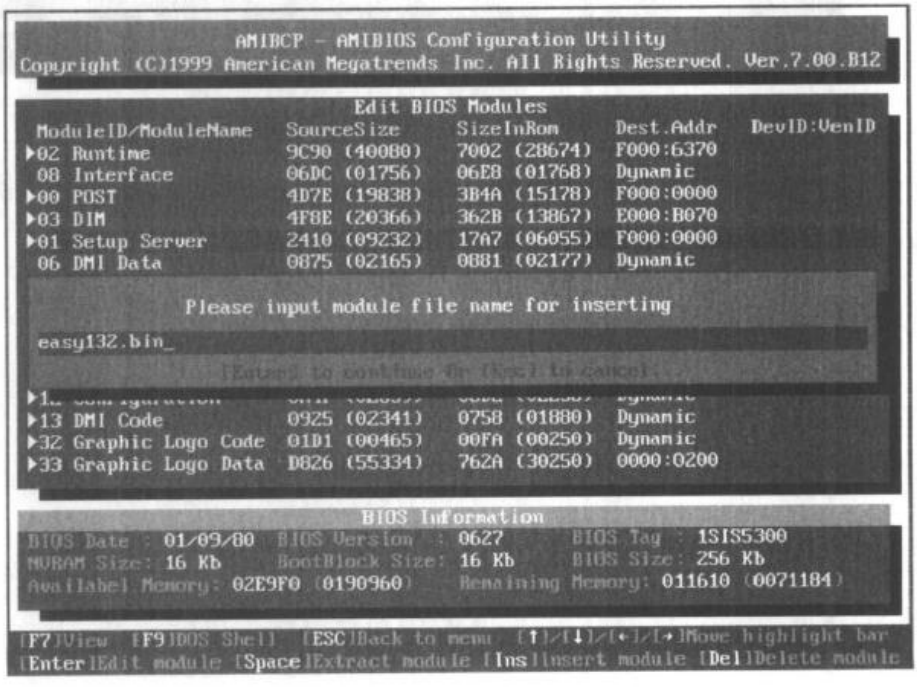


图 10-20　加入新的设置界面文件

接下来会出现要你指定这个模块的 ID、是否压缩以及解压段地址、偏移地址等信息，如图 10-21 所示。

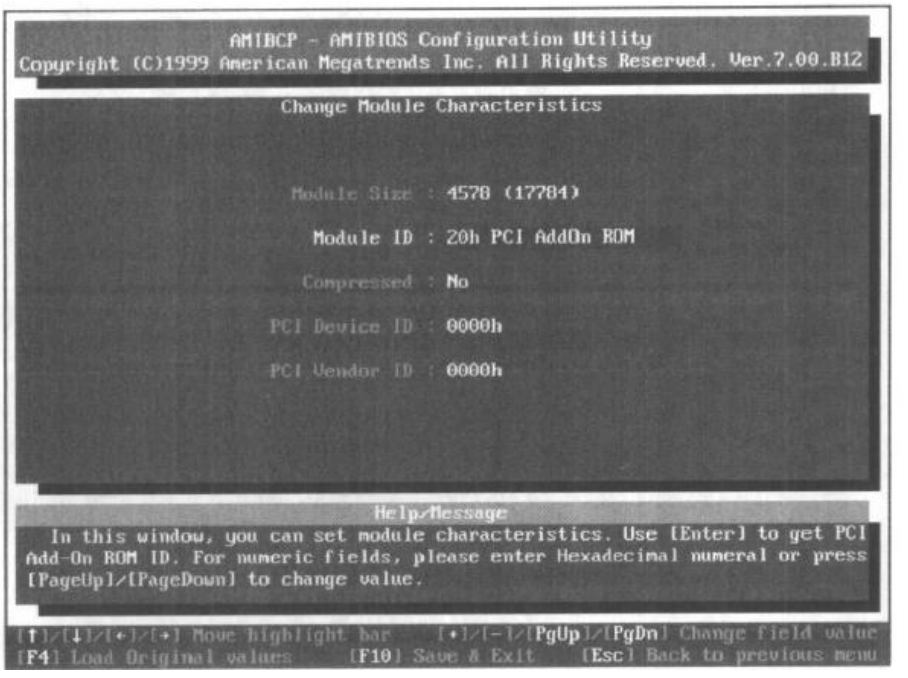


图 10-21　进入指定模块 ID、压缩与解压内存地址

我们指定到 04 Setup Client 并设定压缩，解压内存则保留 0000:0000 (动态解压)即可，如图 10-22 所示。

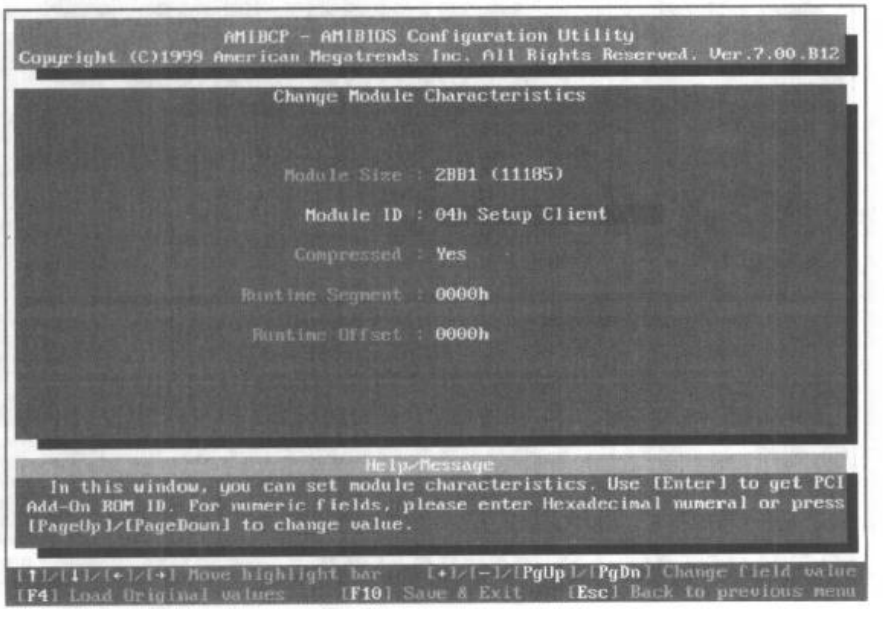


图 10-22　动态解压

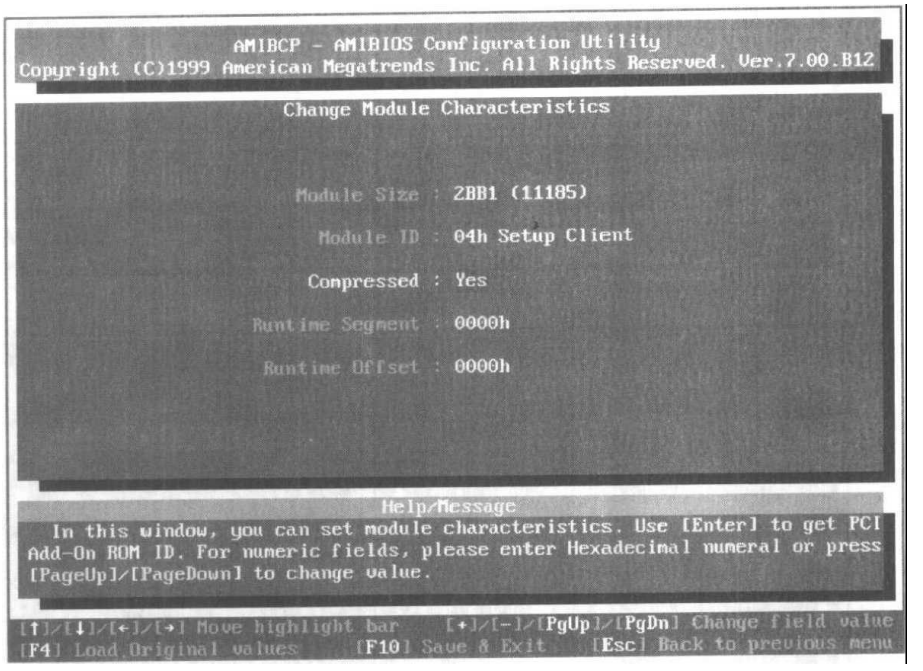


图 10-23　存储新加入的模块信息

接下来按 F10 键存储新加入的模块信息，如果此时贸然按下 Esc 要退回到上一个画面，AMIBCP 也会出现警示信息，提醒是否要将最新更改的模块信息保存起来，如图 10-24 所示。

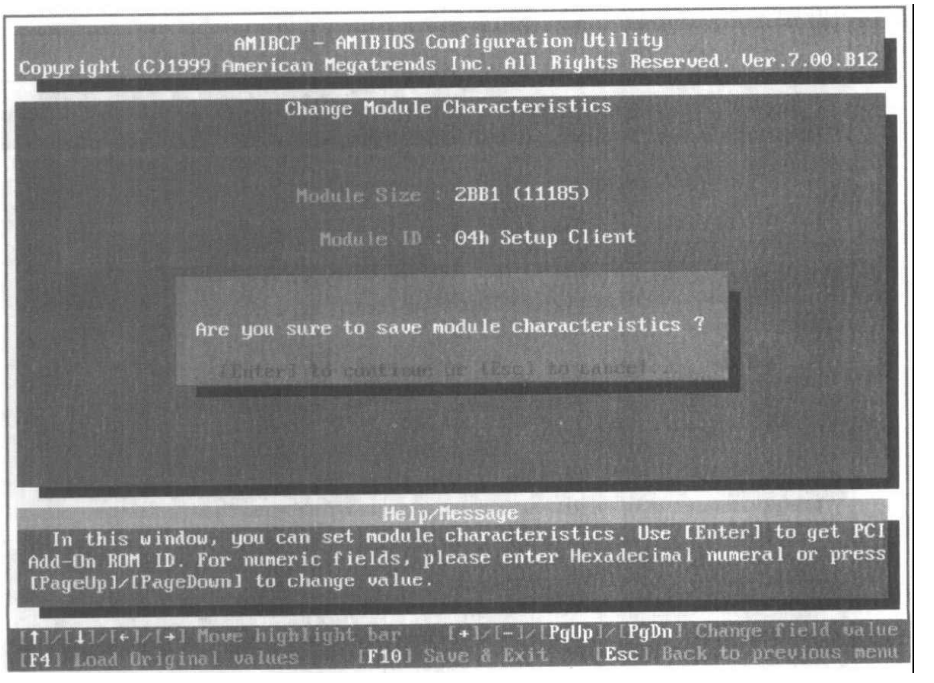


图 10-24　提示保存

保存以后，返回到上一个画面，卷动窗口内的模块项到最后一行，我们会发现这个新加入的 Setup Client 项目。

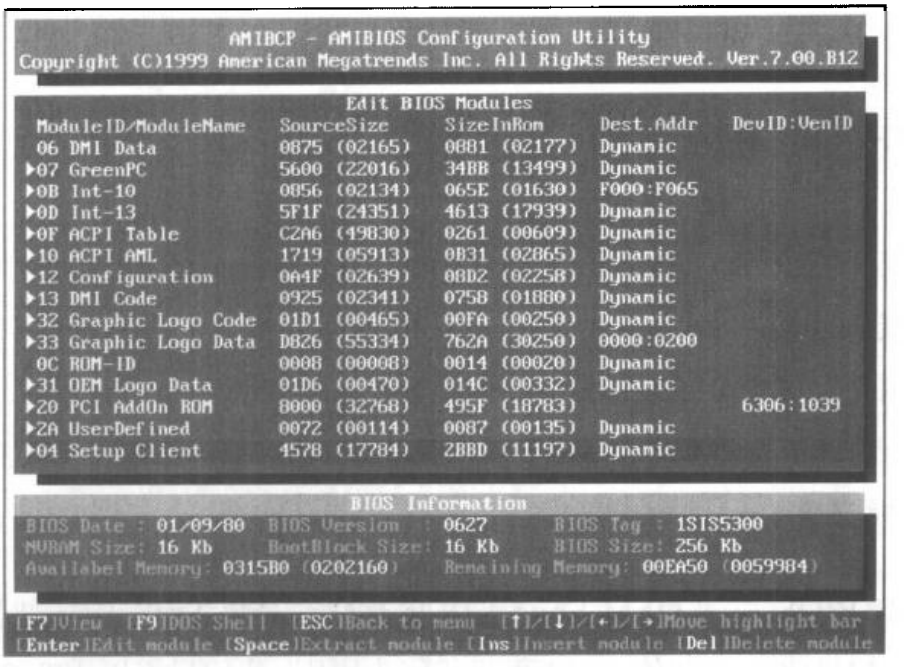


图 10-25　发现新加入的项目

再返回上一层目录，往上选择 Save BIOS to DISK File(将改动过的 BIOS 保存文件)，此时你可以直接以原文件名覆盖，或另取一个新的名称(图 10-26)。

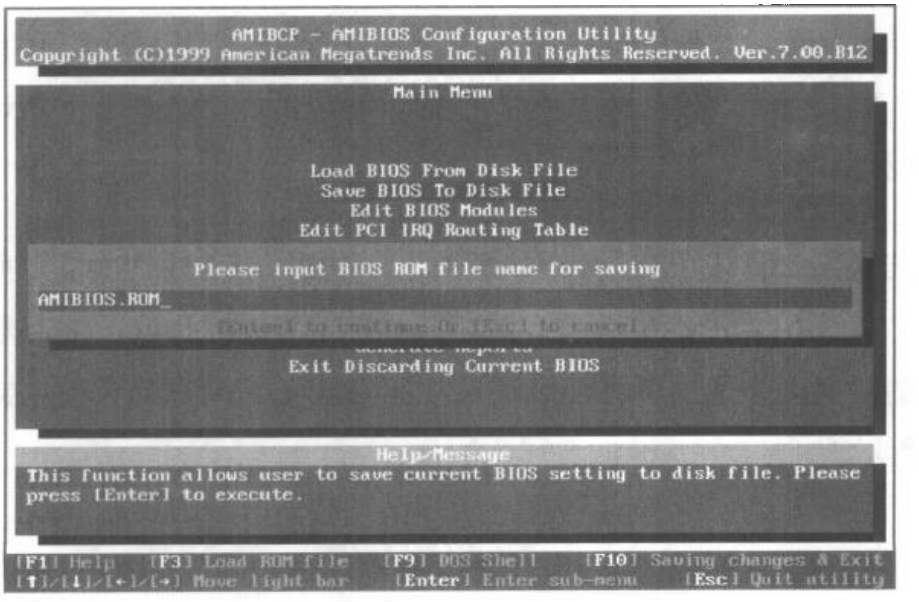


图 10-26　输入保存的文件名

接下来，您只要把这个新的 BIOS ROM 文件烧写到 FlashEPROM 并重新开机，下次您就会看到这个全新面孔的设置画面了。

修改开机的右上角商标图案

在第 9 章介绍了 AMIOLG(AMI OEM Logo Generator)这个应用程序，前面我们也谈到过若以 8×14 或 8×16 文字格式转换，在最多有 96 个可定义字形的使用范围下，这个 BMP 图片的大小就是 128(个水平点)×96(个垂直点)或者 128(个水平点)×84(个垂直点)。接下来，我们就实际制作这样的一个 OEM Logo 图形商标文件。

AMI BIOS 内部 OEM Logo 的程序代码，是以 8x14 的字型为基准，在不更改 AMI BIOS 内部的 OEM 程序代码为原则，我们只好以 128x84 的尺寸大小，作为我们要外挂、扩充的图案尺寸。

我们先找一个准备加入到右上角的图片，源图的分辨率、颜色数不拘，但不要过于复杂，此处我们以 \627.10\SIS530PcOEM\DEMO\Kitty.png 图文件为例子。它是一个 329×298 全彩的图片，笔者是以 Paint Shop Pro 图像处理软件做示范。

首先我们打开这个 kitty.png 图文件，利用区域剪贴的功能，剪出一个长 128 点、宽 96 点的图形区域。此时我们把可爱的猫头取下来，然后利用缩放尺寸的功能，生成一个大小恰好为 128×84 的凯蒂猫猫头图片文件，如图 10-27 所示。

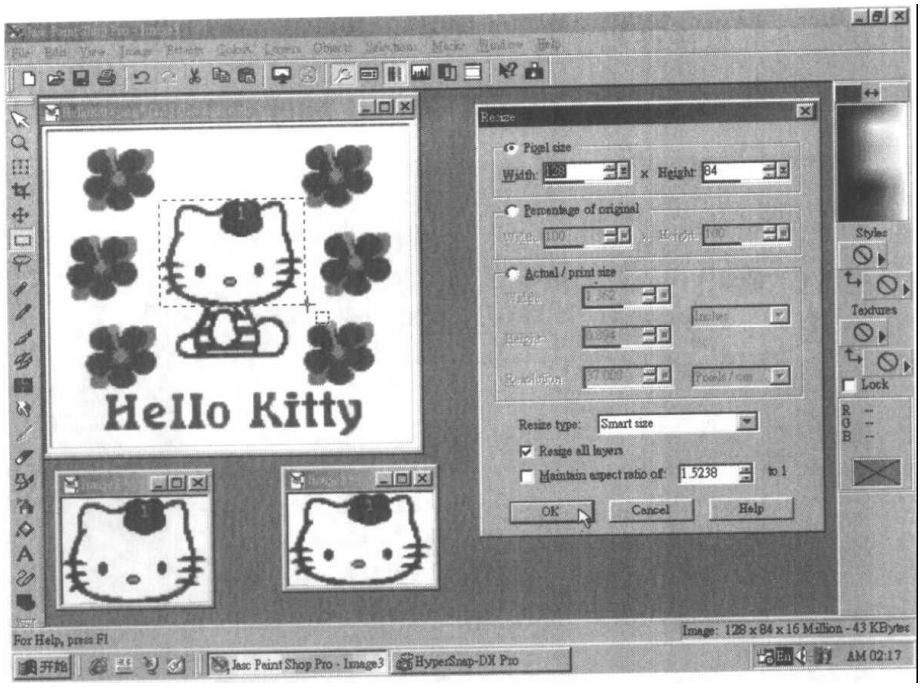


图 10-27　打开 Hello Kitty(hellokitty.png) 商标图片文件，截取猫头并且缩小成 128×84

接下来我们把这个猫头文档用递减颜色到只有两色(黑白)的方式，转成黑白图片，如图 10-28 所示。

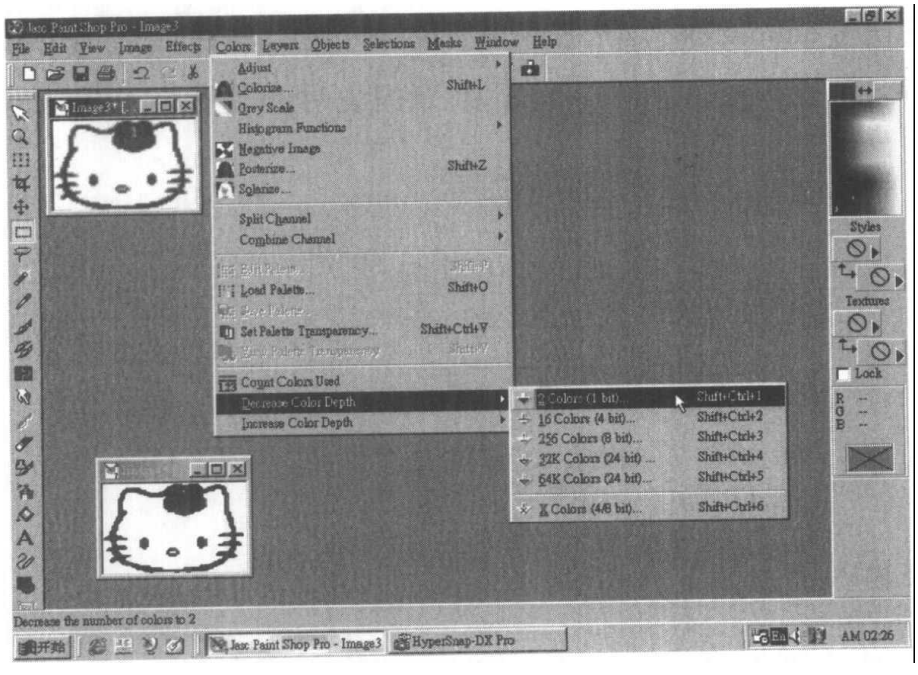


图 10-28　以图像软件转成黑白两色的图片

我们发现凯蒂猫头图样的脸型轮廓线变成黑色，而白色就变成底色，此时我们把它以.bmp 的格式保存起来，取名为 kittywb.bmp ，如图 10-29 所示。

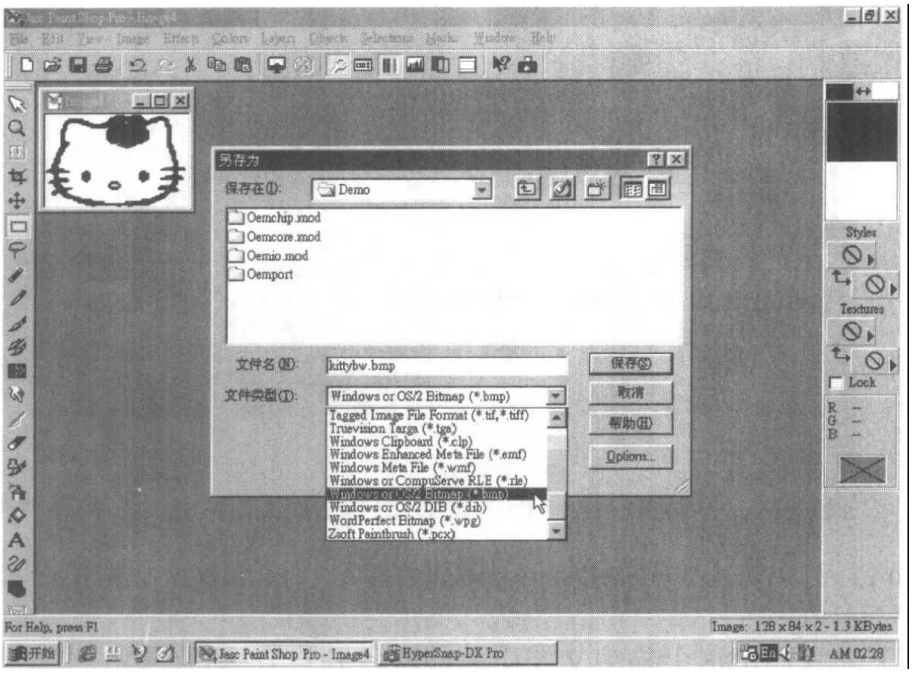


图 10-29　记得保存成.bmp 格式的黑白图片文件

我们使用 amiolg.exe 这个程序来帮我们转换。我们把黑色的轮廓线换成粉红色，背景色设计成白色，参考第 9 章的说明。

闪烁(度)	(红)	(绿)	(蓝)	高亮(度)	(红)	(绿)	(蓝)
F	R	G	B	I	R	G	B
0	1	1	1	1	1	0	0

= 0 7 C h =124(十进制)

白色(背景)　　高亮度红色(即粉红色)(前景)

只要输入下列指令：

```
amiolg /ikittybw.bmp /opinkitty.bin /x64 /y0 /l14 /c124
```

它就会生成一份 pinkitty.bin 的 OEM Logo 图形文件模块(如图 10-30 所示)。

如果我们担心设置的颜色、位置不好，可以输入 amiolg/ipinkitty. bin /d 指定模拟显示。

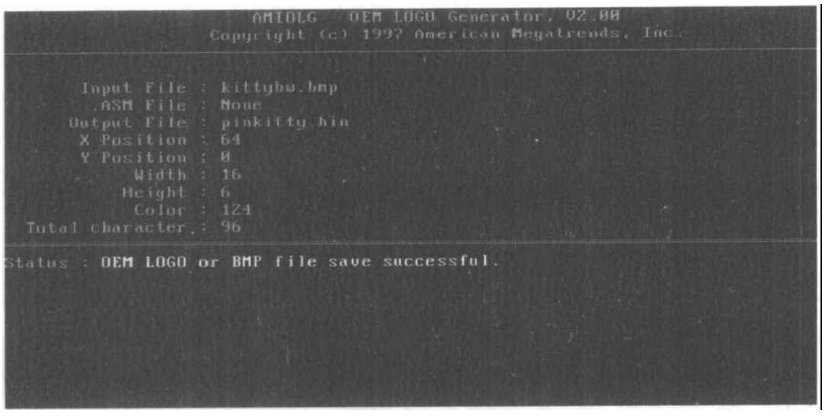


图 10-30　OEM Logo 图形文件模块

一切都测试妥当后，我们同样是运用第 8、9 章介绍的 AMIBCP 组态程序，运用前面介绍过模块删除、加入新模块的方式。

首先用 Flash826(AMIFLASH)将您手边用的主板的 AMI BIOS 保存起来，假设文件名也是 AMIBIOS.ROM 。接下来键入 AMIBCP AMIBIOS.ROM，直接进入 Edit BIOS Modules 选项， 控制光条移到 OEM Logo Data 项目(图 10-31)，然后删除这个模块(如图 10-32 所示)。

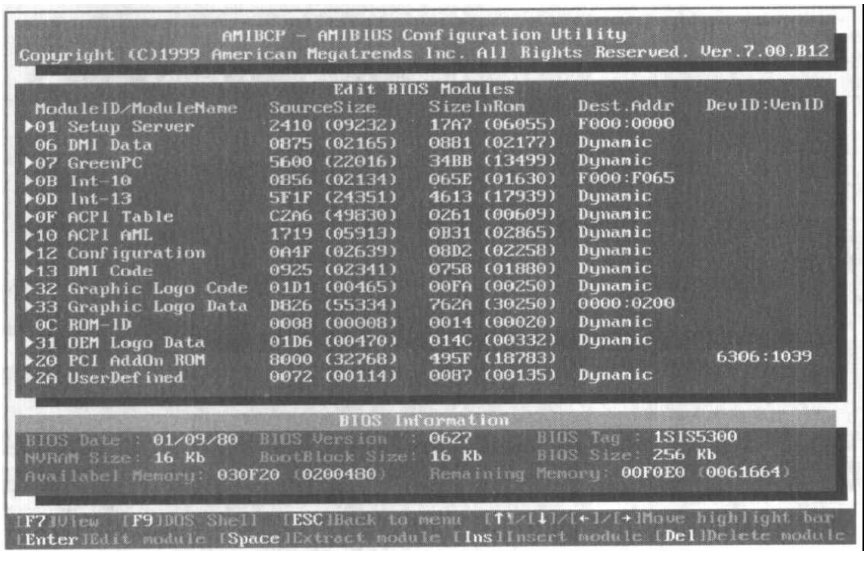


图 10-31　选择 Logo Data 项目

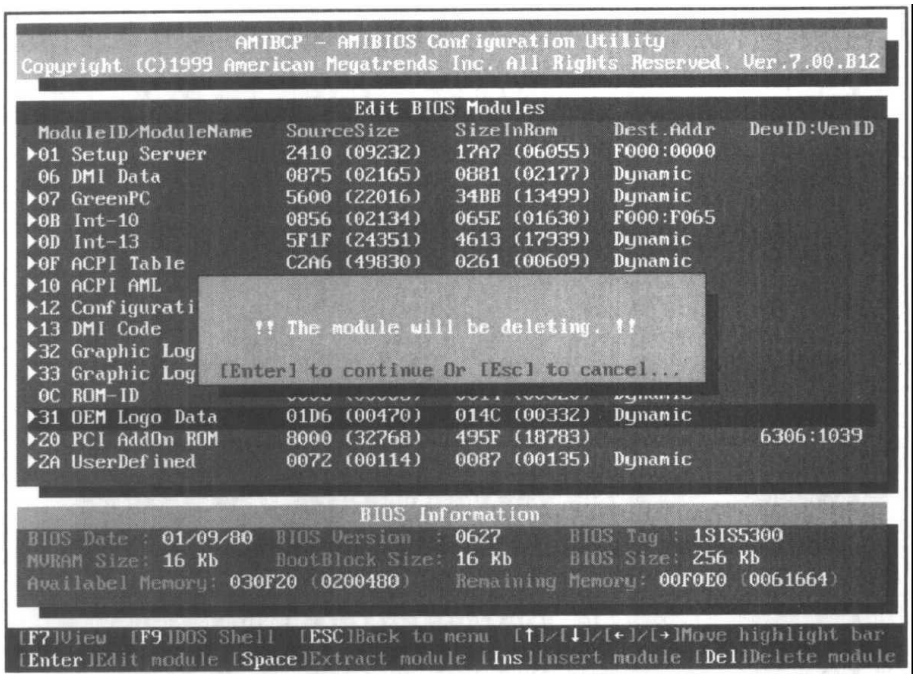


图 10-32　删除这个模块

接下来按下 Insert 键并将 pinkitty.bin 加入到 AMIBIOS.ROM 的模块内，如图 10-33 所示。

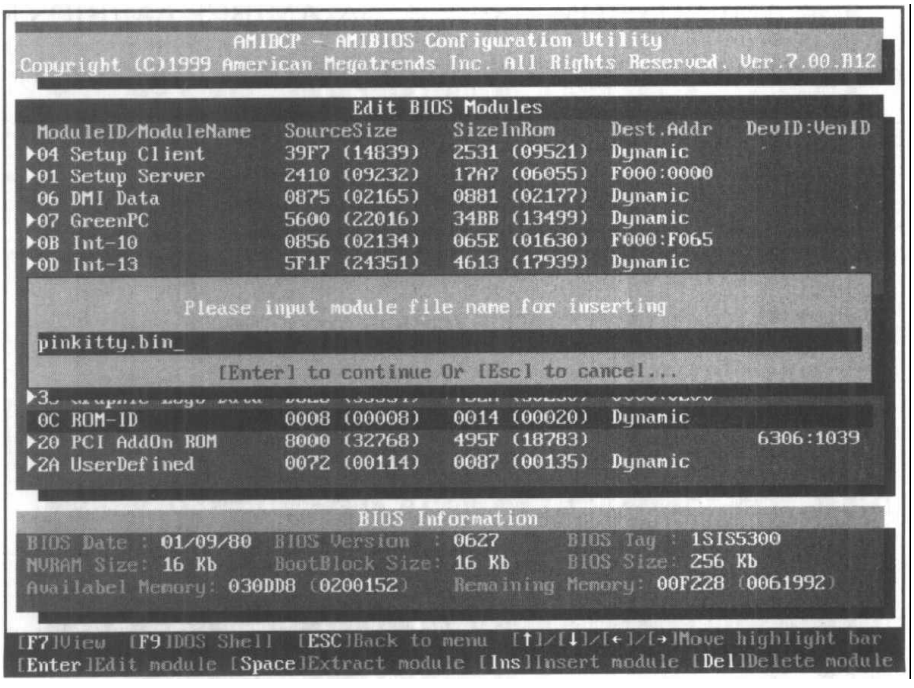


图 10-33　加入这个新的 xxxx.bin 模块

接下来把模块 ID 改成 0Eh，代表 OEM Logo Data，并设定 compressed 为 yes ，解压内存地址都维持 0000h，0000h。

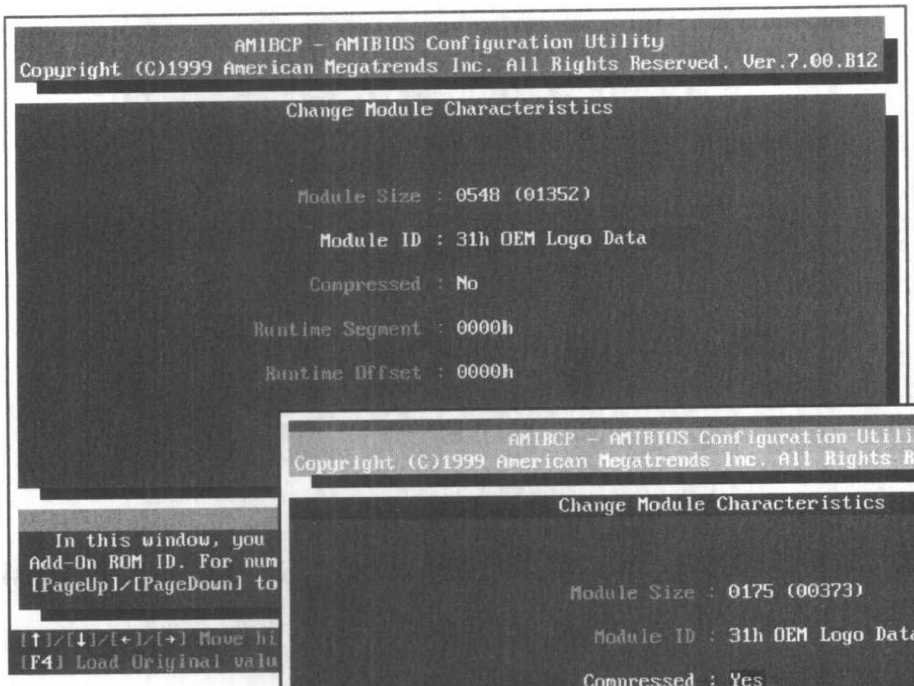


图 10-34 设置

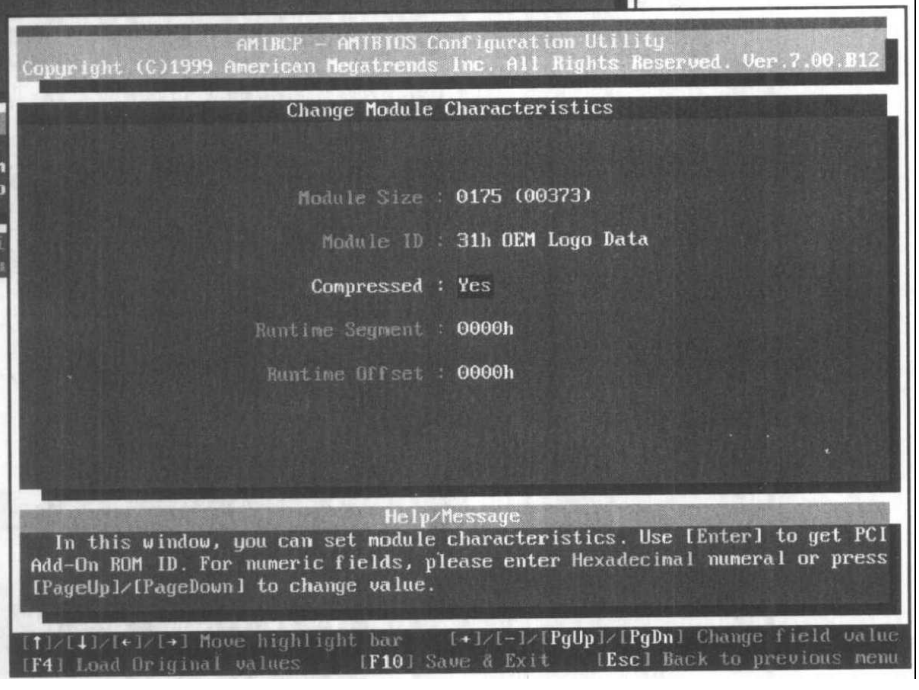


图 10-35 设置

此时按 F10 把新的模块组态保存，然后回到 Edit Modules 窗口，看最后一行是否有一个新加入(Module ID=31h)的模块(图 10-36)。

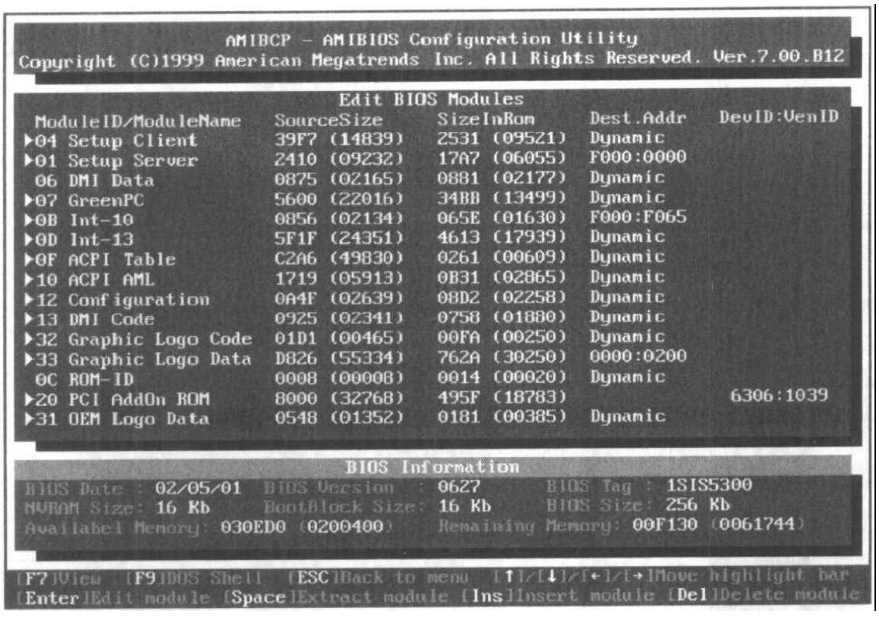


图 10-36　是否有新加入的模块

到目前一切都算正常，此时就可以回到主菜单选择 Save BIOS To Disk File，看是直接保存 AMIBIOS.ROM，或者您另外取个文件名叫 kittyami.rom 也可以。

接下来就打开 Flash826 amibios.rom (或 kittyami.rom)，直接烧进去就行了。图 10-37 是笔者测试成功的画面。

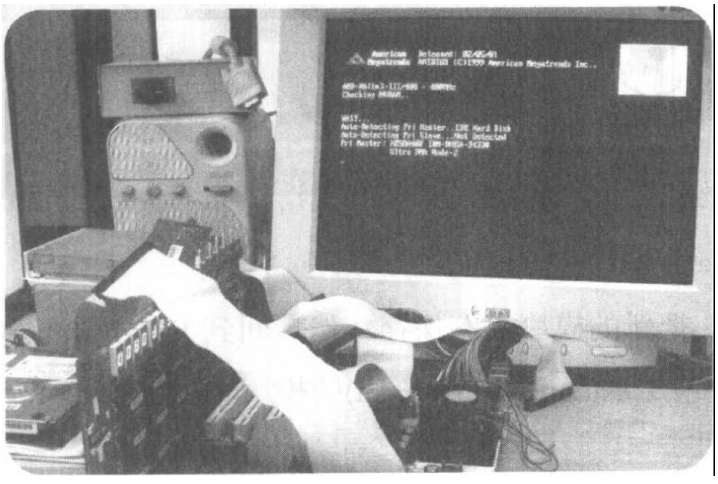


图 10-37　开机时右上角出现粉红白底可爱的凯蒂猫图案(以数码相机实机拍摄)

想追女友的读者，快把这招学起来吧！如果她的的主板也是用 AMI BIOS，只要您显示几招，帮她把 Hello Kitty 加进去(要加别的图案也成)，日后只要您的亲密爱人一启动计算机，右上角就是一只可爱的 Hello Kitty，保证她会爱死您！

加旗帜也可以

咦？既然加图案可以，那么我可不可以也加国旗或其他旗帜呢？

基本上，只要是两色的旗帜，运用一色为背景色(底色)，一色为前景色的原理，都可以显示出最大 128×84、两种颜色的旗帜图案。因为只有两色，都可以运用上述规则，配合 AMIOLG 造出商标图形化字型模块，在右上方显示我所要的旗帜图案。

修改全屏幕图形画面

在第 4 章讲解 OEM.MAK 时，介绍过 SILENT_BOOT 这个字段，以及可指定文件名的 OEM_LOGO_FILENAME，在编译的过程中，如果厂商指定 SILENT_BOOT=1，并且在 OEM_LOGO_FILENAME 这个字段指定全屏幕显示画面的图形文件名，就可以设置为以全屏幕画面静态显示的方式，显示 OEM 厂商特定的全屏幕、全彩图形。

图形文件的长宽分别为水平 640 点、垂直 480 点(640×480)、256 色模式，而图形文件格式为.pcx。

但前面的章节也说过，AMI BIOS 每一个模块的长度不能超过 64KB，而且这是未压缩时的状态。受限于 FlashEPROM 的总容量，以及 BIOS 模块格式的定义限制，AMI 对这个全屏幕图形文件的长度必须有所限制。理论上不能超过 65540(64KB..65536 再扣掉 16bytes)，据笔者的测试结果，最好是文件总长度在 64000~65000bytes 之间，否则可能无法正常显示。

事实上一个 256 色 640×480 的图片，完全不压缩的情况下大小就有 640×480×1bytes =307200bytes=300KB，目前只有买单个 512KB(8Mb) Flash 芯片才装得下去，但以目前大多数主板都还只是 256KB 的情况下，塞下 300KB 的图片也实在太浪费了，因此 AMI BIOS 接受以.PCX 可压缩图形文件格式存储，只要能压缩成为一个小于 64KB 的模块，内建的显示图片程序会自动解压缩这个图文件，并在屏幕显示出来。

如何让 BIOS 在开机时，显示您预先准备好的 256 色全屏幕.PCX 图片呢？

① 取得任何一张您想要摆设的全屏幕画面图文件。

② 打开惯用的图像处理软件，把它转成 640×480 分辨率的图文件。

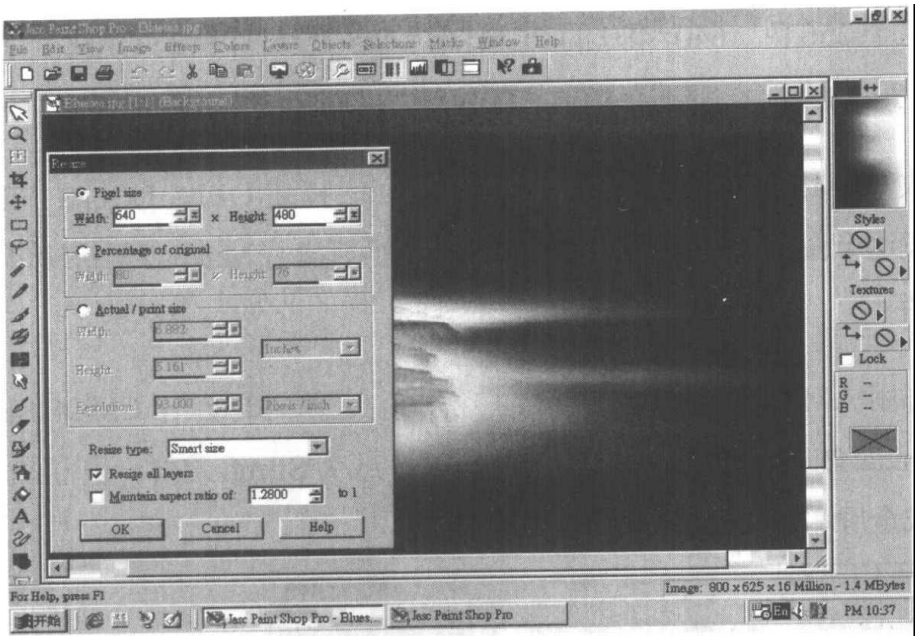


图 10-38　设置分辨率

③ 将已经转成640×480大小的原始图片的颜色数(decrease color depth),递减到只有256色的模式，可以利用图像处理软件的递减颜色的功能 / 选项，多选择几种试试看。

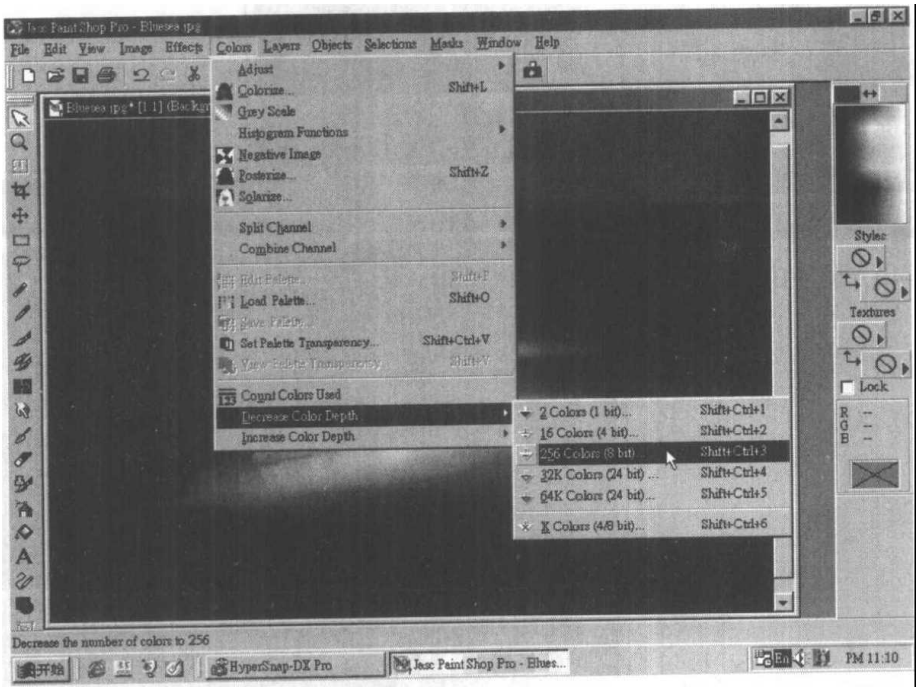


图 10-39　将图片递减颜色到 256 色

④ 将这个256色、640×480的.PCX图片存到您要移植的OEM厂商目录下，确定它的文件长度不会超过64KB。

⑤ 修改OEM.MAK，把Sile_boot设为1，同时指定OEM_LOGO_FILENAME加载这个图文件的完整路径与文件名，然后执行Makeall重新编译监控程序文件。

⑥ 最好用AMIBCP去组态这个监控程序，进入Advanced CMOS Setup项目，把Initial Display Mode的默认值从BIOS改为Silent，这样一开机就确定会切换成全屏幕画面显示模式啦！

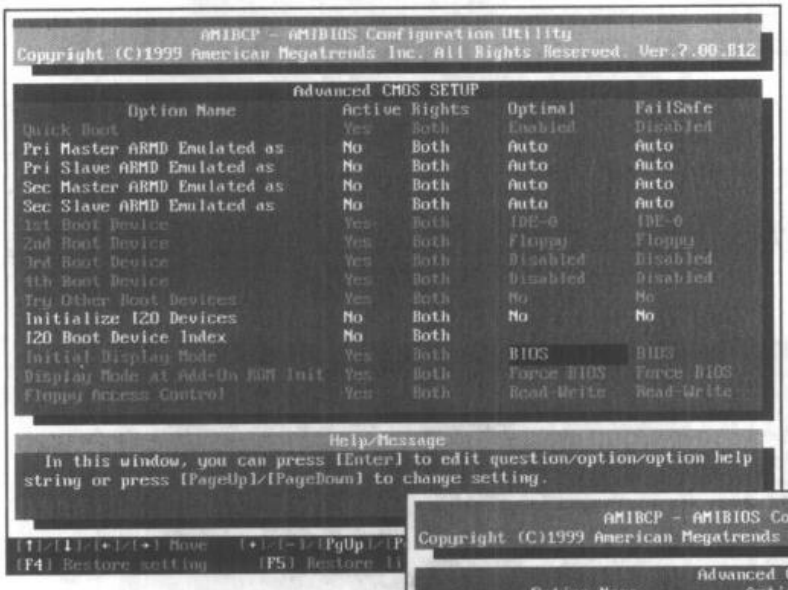


图 10-40 原为BIOS

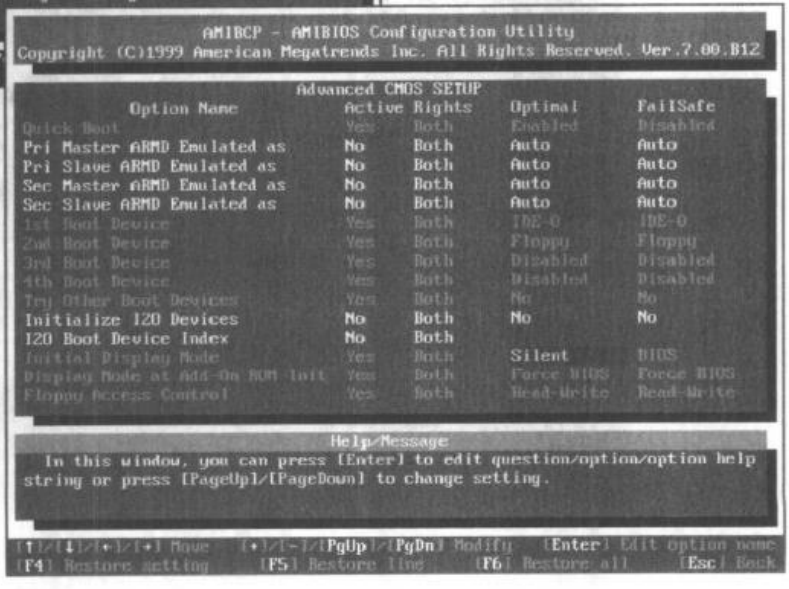


图 10-41 改为Silent

⑦ 参考前面将监控程序烧写到FlashEPROM的方式，以AMI FLASH (Flash826.exe)将编译、组态好的 BIOS 文件烧写进去。

选图的诀窍

由于.PCX 文件的压缩率极低，能压缩到刚好64KB 以下的图片真的很难找。这也就是常见到国外一些公司所设计的台式计算机或笔记本电脑开机时，他们的全屏幕静态显示画面并不会很花哨的原因，往往也只有用色单调的图案背景，配上小区域、不同色的文字、图案，甚至整体偏向同一、两色系都有可能。

可是笔者还是希望有自我风格、自己全屏幕彩色图案开机的 BIOS，那该怎么做呢？在这边提供几个技巧。

① 不要选择画面过分复杂的图片

原则十分笼统，但是选择起来其实很简单，就是选择或创造出有极大片重复景色、区域的图形，而且细节部分变化不要太剧烈，就很容易压缩到64KB 以下。

② 选择相近或相同色系的图片

例如说天空，一大片都是蓝色参杂着一丝丝的白色云彩；另外如静止不动的湖泊、夕阳等都是很好的开机画面取景。

第二点再配合第一点，则更容易成功。

为了找出可让读者实践的图片，笔者决定豁出去了，我好不容易千挑万选了两张图片，原始文件是全彩模式的.jpg 文件，还是 800×600 甚至更高解晰度的图案，自行以图像软件缩图到640×480，并测试几种递减颜色的方式到256色之后，好不容易压到 64KB 以下，就一并附在光盘上供大家采用了。

其中一张是夕阳(sunset.jpg)，整个景偏火红色，而地表与远处有一些黑，另外一张笔者觉得比较好看的是黎明前的蓝海浪涛(bluesea.jpg)。我选择蓝海作为我的全屏幕画面静态展示，以下是笔者实验出来的成果。

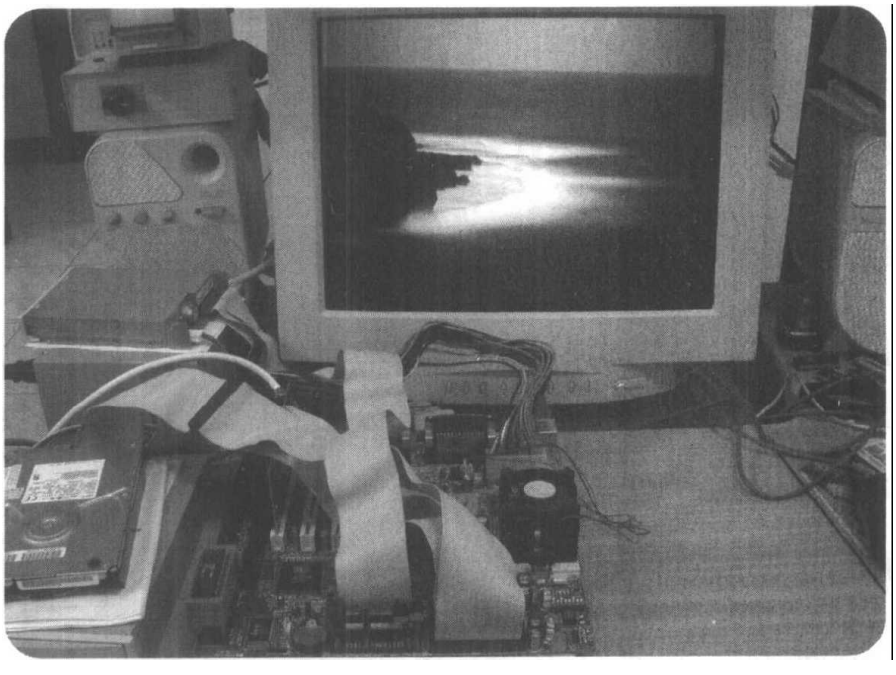


图 10-42　笔者所实验出来的开机全屏幕画面图(蓝海白浪，从实作主板开机画面实际拍摄)

漂亮吧！读者们是否也跃跃欲试了呢？

显示图片的技术限制与题外话

那全屏幕图形静态显示的功能，有没有办法跟上述商标图形模块外挂的做法一样，不需要取得源代码，只要主板上有已经编译好的 BIOS 监控程序文件，也能够自行加工吗？

这一点笔者实际测试过了，你是可以把模块加进去，但如果厂商当初编译时没有设置 silent_boot=1，那么在编译链接时，显示.pcx 图片的模块并不会被载入进来，而且相关标志也没设定好。即使外加了，甚至狠一点，打开 AMIBCP 进行 BIOS 组态，硬是把 Silentboot 选项 active 并设定好，AMI BIOS 一样不会显示图片，内嵌的这个 64KB 的图片就形同垃圾一般。

除非一种情况，就是你取得的 BIOS 本身就有全屏幕图形显示的功能，那在这种情况下，厂商当初编译 / 链接时已经设好相关的标志，显示图片的模块也就进来了，那么外挂模块的做法，相当于把厂商预设的全屏幕图形模块给“替换掉”，在这种条件下是可以的。

最后，也希望 AMI 程序设计小组若行有余力，能早日加入可解压 / 显示.jpg 图形的汇编语言码，因为.JPG 图形文件支持全彩模式，而且压缩率比较高(甚至可高达 1/16)，而人眼无法瞧出这细微的失真。那么一张未压缩却动辄 800、900KB 甚至 1MB 的图片，只要被压缩在 64KB 以内，也是有可能加到 ROM 的。

当然我知道这不是一件容易的事情，除了显示.JPG 图片的汇编语言码真的不好写外 (又不能拿 C 语言去转，这是因为 C 语言有许多链接库，而这些链接库又要调用 BIOS 甚至 DOS，微缩程序代码不是件容易的事)，还要考虑到主板的 FlashEPROM 容量，够不够装下这些庞然大物，想了就令人头疼啊！

但是为了让计算机开机显示更高画质、更丰富色彩的图形，一直都是笔者所衷心期盼的，也希望有汇编语言潜力的读者，大家来跟笔者一同努力吧！

10-5 BIOS 开发的错误解决与明日应用

强大的 AMI Diagnostic 除错卡

发展 BIOS，如果手边能有些强大的神兵利器，在千头万绪的 BIOS 程序代码中找寻错误并迅速修改，当然会比较容易一点。

在笔者向 AMI 公司取经的过程中，也看到 AMI 公司早年发展了一块特殊的诊断除错卡，一张市价一万多元，如图 10-43 所示。

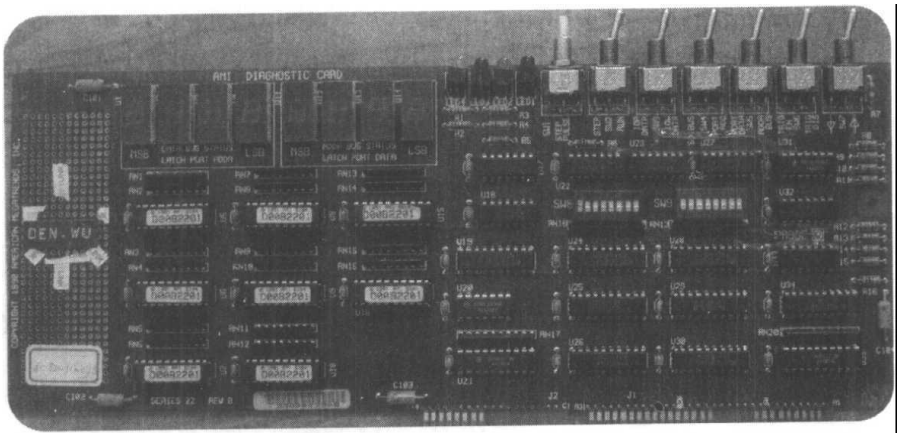


图 10-43 AMI DIAGNOSTIC Card

这张卡是传统 ISA Bus 接口设计的老卡，不过功能可比一般简易的 Debug Card (利用 port 80h 送出给 Latch)强太多了，它可以做显示除错值、执行地址、IO 端口读写状态的检测，还可以做暂停、单步执行的动作。图 10-44 是这张卡配合移植端主板工作的情形。

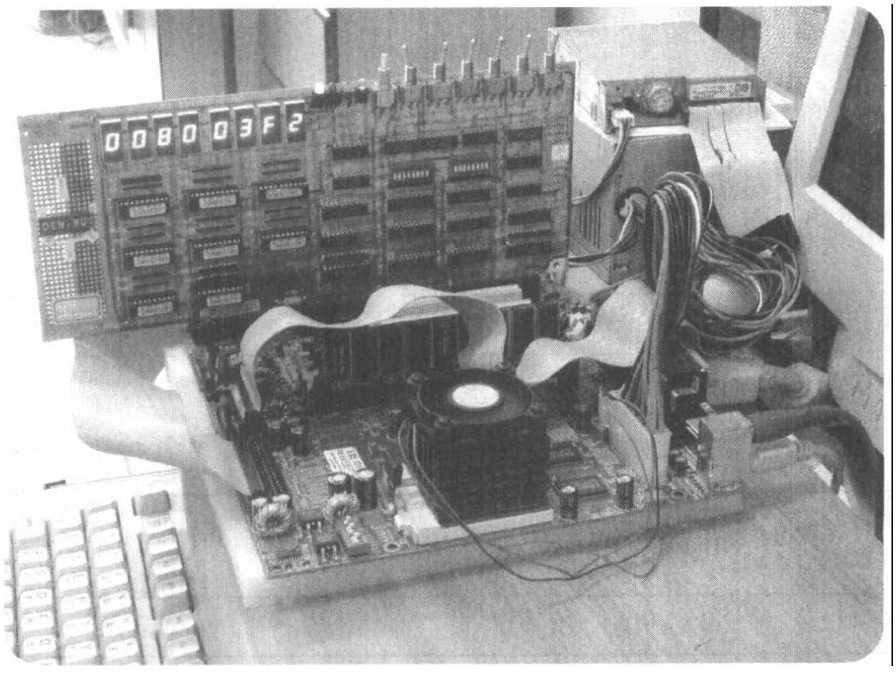


图 10-44　AMI DIAGNOSTIC Card 与主板搭配除错情形

笔者拍摄开机画面也是借助这张神兵利器的暂停执行的功能，才有办法调整相机来拍摄，要不然每个开机画面闪一下就过去了，根本没办法有足够的时间去抓住那瞬间，并转换为永恒的画面。

当然啦! 读者只是初步对开发 BIOS 有兴趣，也才刚从 BIOS 的编译、PCI IRQ Routing 与模块外挂的奋战中初尝艰辛的战果，除非您就准备一头栽到这个行业中，要不然这类神兵利器暂时还用不着。

除错码(POST/Debug Code) 所涵盖的源代码

在本书章节即将迈向完结的阶段，我们整理了一份 AMI BIOS 诊断码所涵盖的源代码对照表，希望对读者以及专业人员研发、除错 BIOS 时，能有一些帮助。

源代码文件名	牵扯到的 POST/Debug Code(诊断码)
UB.ASM	D0、D1、D3、D4、D5、D6、D7、 E0、ED、EE、EF、F0、F1、F2、F3、F5(BIOS 自我烧录过程)
MEMORY.ASM/ OEMMEMRY.ASM	D3
BIOS.ASM	03、05、06
RUNTIME.ASM	05、B1
OEMRUN.ASM	05
RM.ASM	07、08、0B、0C、0E、0F、10、11、12、13、14、 19、1A、23、24、25、27、28、29、2A、2B、2C、 2D、2E、2F、30、31、32、33、34、37、38、39、3A
NPOST.ASM	0B、27、37、39、50、88、8C
POST.ASM	13、57、8C
OEMPOST.ASM	0E
DBUG.ASM	2A
RUNACPI.ASM	B1
SETUP.ASM/ OEMSETUP.ASM	87

从本书第3章第三节 BIOS 的诊断码，以及上表的对照关系，再配合第5章BIOS的内部模块与执行流程介绍，大致找出可能涉及的源程序代码(*.ASM)再去做修改，相信在做 Porting & Maintaining BIOS 时，对您会有很大的帮助!

如果您还是有一些技术上的问题与疑惑，除了和业内的 BIOS 研发/维护工程师们互相讨论求解之外，或者向芯片组的厂商、原生 BIOS 开发厂商(此处当然是指 AMI 公司啦!)寻求技术上的帮助，当然您也可以写封 email 给笔者，笔者很愿意就自己的能力所及做回答，或者代为向 BIOS 厂商询问。

笔者的 email 信箱是 chenbros@gcn.net.tw 。

BIOS 在明日的应用

至于 BIOS 还会有些什么新的应用呢?

笔者自己在业界的秘密探访中发现下列几项会是新兴应用的功能，其中有几项已经有厂商开发出来，为自己的产品添加某些特殊功能，甚至还去申请专利。

① 近乎自动化的外频/倍频线上调整功能，兼超频当机侦测与还原回最低速的功能。

原理：第4章Boot Block程序代码自行改写，配合第7章程序调整值的极佳表现，硬件上则配合Watchdog timer for reset(计时reset线路)。

实验成功厂家：升技(ABIT)、华硕(ASUS)

② BIOS通过Windows上网网络直接联机，Download & Flashing

原理：自行掌握Flash烧写程序代码技术，设好调用点后，在窗口中用应用程序将其调用进来，需要BIOS程序代码与Windows应用程序互相配合。

实验成功厂家：华硕(ASUS)、技嘉(GigaByte)、微星(MSI)

③ Dual BIOS双重BIOS自动侦测/还原

原理：同样掌握第4章Boot Block、Flashing以及其他相关的技术，配合硬件watchdog timer计时切换线路，由Boot Block侦测主BIOS与备用BIOS芯片的内容。

实验成功厂家：技嘉(GigaByte)、承启(Chaintech)

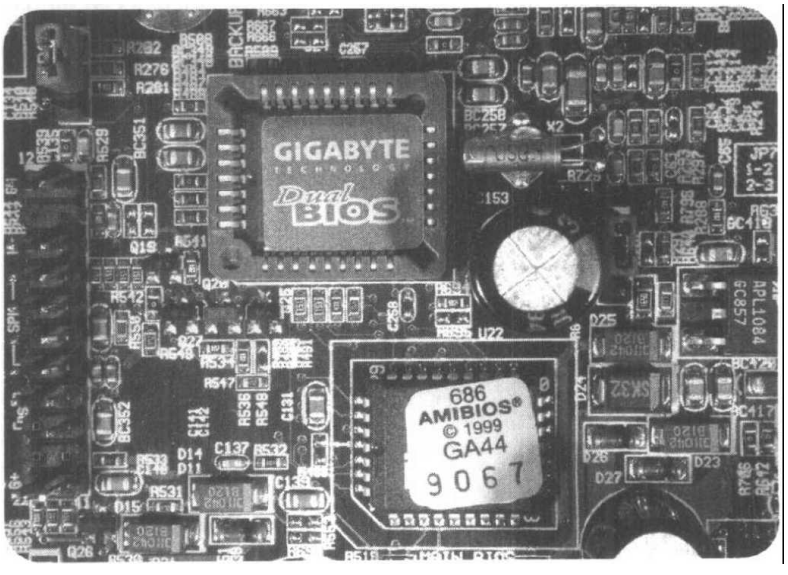


图10-45 技嘉的Dual BIOS功能

④ 语音警示信息

原理：配合预先录制的语音 IC，并修改 BIOS 的 POST Code，摘录一些使用者接触到的 Post Code，以 GP I/O 接口从语音 ROM 芯片中提取波型文件，直接送喇叭输出。

实验成功厂家：伟格(Soltek)、大众(FIC)、微星(MSI)

⑤ 防毒软件、硬盘修护软件... 防黑客软件等等

原理：利用附加模块的原理，在 BIOS 开机阶段将其解压开来，用来拦截相关的磁盘中断向量的小程序。

⑥ BIOS 嵌入式操作系统(Linux) 以及网络浏览器

像硅统开发出能在 Disk-On Chip(一种仿真磁盘驱动器的大容量 Flash 芯片)装置，安装其修改过的 Linux 操作系统，整个系统可以一开机就快速加载 linux 系统。

威盛公司更是和旗下投资子公司 Elegant 合作开发了 EtBIOS，短短 256KB 的容量，可以把类似 Internet Explorer(或 Netscape Netvigator)功能的浏览器程序内置进来，计算机只要一开机就加载整个浏览器，可以直接上网浏览、写信，这对于目前方兴未艾的 IA 装置的研发，是一项体积小、成本低廉而且富有竞争力的产品。

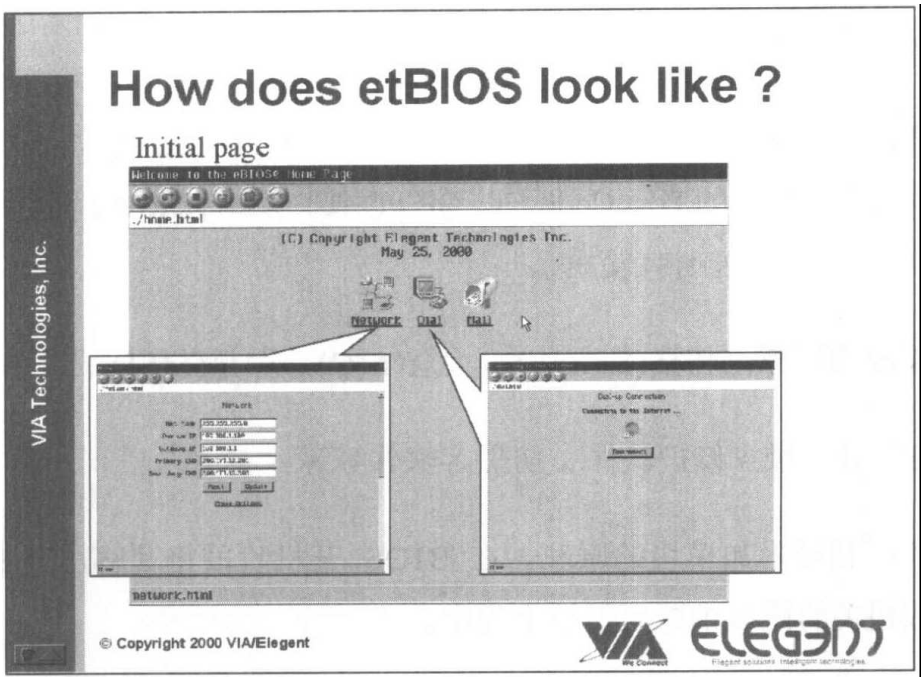


图 10-46　BIOS 威盛的 etBIOS 程序画面

现在源代码就掌握在您的手上，您想增加些什么新的应用呢？大家一起动动脑筋吧！

A

附录

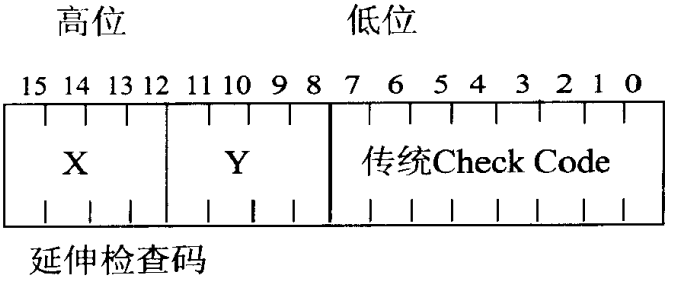
A-1 AMI BIOS的POST错误信息代码表补注

第3章中我们曾经介绍过AMI BIOS的POST/Debug Code。一般POST/Debug Code只对Port 80h送出一位的除错码，但在下列情况下，则会送出16位的除错码。

除错检查码	检查点定义叙述
2A	Different BUSes init (system,static,output devices) to start if present. （介于系统、固定或输出装置的特定总线冲突）
38	Different BUSes init (input,IPL,general devices) to start if present. (介于输入装置、启动装置与一般外围装置的特定总线冲突)
39	Display different BUSes initialization error messages. 特定总线的装置初始化错误，并显示错误信息。
95	Init of different BUSes optional ROMs from C800 to start. 从特定总线附加ROM区域读取/执行程序码，C800段地址到目前的地址产生冲突情况。

在此处Different Bus（特定总线）指的是主板PCI Bus之外的另外一组Bus；对于常见的台式计算机主板来说，指的就是16bit AT-Bus（针对Super I/O芯片）；对于Intel系列主板来说，指的是符合LPC（Low Pin Count）接口的总线。

所以当读者看到这些除错码时，由于此时AMI BIOS还送出一组8bit高位元组（HighByte），这就需要较特别的除错卡才能显示/拦截（例如第10章介绍的AMI Diagnostic卡），底下是这个高位元组的特别定义：



若把高位又分成两个nibble（一个nibble为4个bit），较高位置的nibble为X，较低位置的nibble为Y，那么X的数值可以从0—7（0000b—0111b），其定义如下：

X=0 = func#0,disable all devices on the BUS concerned.
(开关所有位于特定总线上的装置)
X=1 = func#1,static devices init on the BUS concerned.
(位于特定总线的固态形装置/芯片初始化的问题)
X=2 = func#2,output devices init on the BUS concerned.
(位于特定总线的输出装置/芯片初始化的问题)
X=3 = func#3,input devices init on the BUS concerned.
(位于特定总线的输入装置/芯片初始化的问题)
X=4 = func#4,IPL devices init on the BUS concerned.
(连接特定总线的可启动外围装置的初始化的问题)
X=5 = func#5,input devices init on the BUS concerned.
(连接特定总线的其他装置的初始化的问题)
X=6 = func#6,input devices init on the BUS concerned.
(回报相关的总线错误)
X=7 = func#7,input devices init on the BUS concerned.
(附加卡ROM内存区域相冲突，像内存地址重叠...)

至于低位 nibble Y，它的数值可以从 0 — 5（0000b — 0101b），其定义如下：

Y=0 = Generic DIM(Device initialization Manager 装置初始化管理者).	
Y=1 = On-board System device.	(焊在主板上的芯片/装置)
Y=2 = ISA devices.	(ISA 接口设计的芯片/装置)
Y=3 = EISA devices.	(EISA 接口设计的芯片/装置)
Y=4 = ISA PnP devices.	(ISA 接口设计,符合 PnP 规格的芯片/装置)
Y=5 = EISA devices.	(PCI 接口设计的芯片/装置)

A-2 AMI Beep 错误声音代码表

如何以喇叭声音来判断系统错误源

当系统打开电源开关后，有时系统/主板无法显示任何信息，却又陷入停滞状态，若没有其他主板、装置来交叉测试，甚至动用到测量仪器，有时很难判断问题所在。AMI BIOS 也针对一些错误状况，发出不同的声音组合来提醒使用者。

这些供除错用的特定声音意义，其说明如下。

连续发声的次数	错误代表的意义（信息中文翻译）	错误原因
1	DRAM refresh is not working. （DRAM 刷新功能不正常）	内存故障、DIMM 插槽故障或芯片组的问题
2	Parity circuit is not working or parity bits are not cleared when parity is disabled. （同步位检查电路异常或状态位无法清楚）	内存Parity位问题 BIOS设置或芯片组问题
3	First 64k memory test failure.Address line test (A0...A15) failure. （第一个 64K 内存区测试错误）	内存故障或芯片组问题
4	System timer is not counting properly. （系统计时器无法计时）	Super I/O或南桥芯片故障
5	Processor register/flag test failure. （处理器寄存器 / 标志位测试错误）	CPU 故障，或主板、总线电路设计、搭配上有问题
6	8042 keyboard controller gate-A20 erroe. （8042 键盘控制器的 A20 控制电路错误）	Super I/O或南桥芯片故障
7	Processer exception error encountered. （处理器例外错误）	CPU 故障，或主板、总线电路设计、搭配上有问题
8	Display memory R/W test failure (NON-FATAL) 显卡所使用的内存测试错误 （并非严重到无法开机）	显示芯片、内置显示功能的北桥芯片或显示内存错误
9	ROM-BIOS checksum error. （ROM BIOS加总查核值错误）	FlashEPROM芯片故障，或烧写错误的BIOS监控程序
10	CMOS Shutdown Reigister R/W error. （COMS 当机记录内存读写错误）	RTC错误，属于Super I/O或南桥芯片
11	beeps Cache memory bad. （主板或 CPU Cache 快速存取故障）	CPU、主板北桥芯片或快速存取存储器芯片故障

A-3 相关参考书目

1. AMI BIOS98 User Guide .....Date: Nov 07,1997

2. AMI BIOS98 Porting Guide .....Date: Nov 13,1998

3. AMI BIOS98 Technical Reference.....Date: May 01,1998

4. SiS530 Host,PCI,3D Graphics&Memory.....Date: Nov 10,1998

5. 98DIY3 电脑升级与扩充 <旗标发行>.....Date: Oct 01,1997

6. 98DIY4 电脑新知新选购 <旗标发行>.....Date: May 01,1998

[General Information]
书名=BIOS研发技术剖析
作者=陈文钦著
页数=488
SS号=10454750
出版日期=2001年09月第1版

封面页
书名页
版权页
前言页
目录页
第1章　BIOS基本概论与设置
1-1　BIOS的定义与解释
1-2　何谓BIOS设置、如何更改CMOS设置
1-3　BIOS市场风云录
1-4　简介美国安迈(AMI, American Megatrends Inc.)公司
第2章　硬件基本概论与规格导读
2-1　计算机处理器、主板、芯片组的介绍
2-2　主板零组件、总线等特定规格介绍
2-3　跟电源管理有关的规格与术语解释
第3章　BIOS开发与硬件之间的互动关系
3-1　CPU历代演变与BIOS的关系

3-2　x86CPU内存结构与ROM大小的演进
3-3　BIOS的诊断码(Post Code)简介
3-4　BIOS所要掌握的系统资源
3-5　CMOS的索引数据
第4章　AMI源代码目录结构与文件维护
4-1　AMI BIOS的源代码版本缘由
4-2　AMI BIOS的源代码目录设计
4-3　编译BIOS所需要的编译/链接程序
4-4　BIOS的编译/维护作业的相关文件之一(IO.MAK)
4-5　BIOS的编译/维护作业的相关文件之二——供OEM厂商自定规格/功能等设置值的维护文件OEM.MAK
第5章　BIOS的内部模块与执行流程介绍
5-1　BIOS内部模块的说明

5-2　BIOS执行流程图
5-3　BIOS程序源代码浏览(一)——BootBlock(激活区块)的相关模块
5-4　BIOS程序源代码浏览(二)——Flashpart(快速存取存储器)相关程序代码分析
5-5　BIOS程序源代码浏览(三)——其他在POST阶段参考到的程序代码分析
第6章　BIOS平台移植的关键技术——特定平台修改与关键模块/源代码的介绍
6-1　掌握芯片组、外围I/O芯片的相关规格
6-2　移植到特定芯片组要修改的源代码
6-3　内存容量检测/计算
6-4　PCI IRQ号码配置与绕线设置
第7章　BIOS画面选项默认值设计
7-1　AMI BIOS的开机设置画面

与选项简介
7-2 以AMIBCP调整开机默认值/隐蔽项目
7-3 AMI BIOS的Setup Script (.SSP)语言的介绍
7-4 直接取用源代码文件群的.SSP来修改
第8章 AMI BIOS工具程序——ROM升级与编辑工具
8-1 AMI BIOS基本工具程序列表
8-2 先从烧写BIOS开始——强大的AMIFLASH ROM烧写程序
8-3 BIOS组态魔术师——AMIBCP.EXE之一
8-4 BIOS组态魔术师——AMIBCP.EXE之二
第9章 AMI BIOS工具程序——研发/维护工具
9-1 BIOS组态魔术师——AMIBCP.EXE之三
9-2 AMIMM.EXE BIOS模块操作

/修改程序
9-3　AMIMM.EXE的描述/操作语言介绍
9-4　RU.EXE硬件系统寄存器检视程序
9-5　其他研发工程师/高等玩家的工具程序
第10章　硬件移植实战DIY
10-1　简介本书所示范的主板与其他配备
10-2　建立必要的目录、相关文件与路径
10-3　开始进行Porting作业
10-4　修改开机的设置界面、商标图案与全屏幕画面
10-5　BIOS开发的错误解决与明日应用
附录A
附录页